TESIS DOCTORAL

MODELO DE CONCURRENCIA DE ADA: IMPLEMENTACIÓN Y SUS IMPLICACIONES EN EL INTERFAZ CON EL ENTORNO

que se presenta en la

FACULTAD DE INFORMÁTICA

para la obtención del grado de

DOCTOR EN INFORMÁTICA

AUTOR: **Fernando Pérez Costoya**
Licenciado en Informática

DIRECTOR: **Antonio Pérez Ambite**
Profesor Titular de Arquitectura y Tecnología de Computadores

Madrid, Octubre de 1991

TESIS DOCTORAL

MODELO DE CONCURRENCIA DE ADA: IMPLEMENTACIÓN Y SUS IMPLICACIONES EN EL INTERFAZ CON EL ENTORNO

TRIBUNAL CALIFICADOR

Presidente: D. Pedro de Miguel Anasagasti

Vocales: D. Enrique Mandado Pérez

D. Mateo Valero Cortés

D. Sebastián Dormido Bencomo

D. Angel Álvarez Rodríguez

A mis padres y, especialmente,

a mi hermana Silvia por su inestimable colaboración

AGRADECIMIENTOS

Me gustaría dar las gracias a mis compañeros del Departamento de Arquitectura y Tecnología de Sistemas Informáticos por su colaboración en el desarrollo de este trabajo.

RESUMEN

El principal problema que impide actualmente una mayor utilización de las máquinas paralelas es la falta de herramientas de programación que permitan generar programas transportables a máquinas con diferentes prestaciones. En este trabajo se ha estudiado si los lenguajes con paralelismo explícito cumplen este requisito y son, por lo tanto, adecuados para programar este tipo de máquinas. El exceso de paralelismo, esto es, el uso de mayor paralelismo en el programa que el proporcionado por la máquina para esconder la latencia en la comunicación, se presenta en este trabajo como una solución a los problemas de eficiencia de los programas con paralelismo explícito cuando se ejecutan en máquinas que no tienen una granularidad adecuada. Con esta técnica, por lo tanto, los programas escritos con estos lenguajes pueden transportarse con eficiencia a diferentes máquinas.

Para llevar a cabo el estudio de los lenguajes con paralelismo explícito, se ha desarrollado un modelo abstracto de paralelismo, en el cual un sistema está formado por una jerarquía de máquinas virtuales paralelas. Este modelo permite realizar un análisis genérico de la implementación de este tipo de lenguajes, ya sea sobre una máquina con sistema operativo o directamente sobre la máquina física.

Este análisis genérico se ha aplicado a un lenguaje de este tipo, el lenguaje Ada. Se han estudiado las características específicas de Ada que pueden influir en la implementación eficiente del lenguaje, analizando también la propuesta de modificación del lenguaje correspondiente al proceso de revisión Ada 9X.

Dentro del marco del modelo de paralelismo, se analiza también la problemática específica de las implementaciones del lenguaje sobre el sistema operativo. En este tipo de implementaciones, las interacciones de un programa con el entorno externo pueden causar ciertos problemas, como el bloqueo del proceso correspondiente del sistema operativo, que disminuyen el rendimiento del programa. Se analizan estos problemas y se proponen soluciones a los mismos. Se desarrolla en profundidad un ejemplo de este tipo de problemas: El acceso al estándar gráfico GKS desde Ada.

ABSTRACT

The major obstacle to the widespread utilization of the parallel machines is the lack of programming tools allowing the development of software portable between machines with different performance.

This dissertation analyzes whether languages with explicit parallelism fulfil this requirement. The approach of using programs with more parallelism than available on the machine (parallel slackness) is presented. This technique can solve the efficiency problems appearing in the execution of programs with explicit parallelism over machines with a too coarse granularity. Therefore, with this approach programs can run efficiently on different machines.

A new abstract model of parallelism allowing the generic study of the implementation of languages with explicit parallelism is developed. In this model, a parallel system is described by a hierarchy of parallel virtual machines.

This generic analysis is applied to Ada language. Ada specific features with problematic implementation are identified and analyzed. The change proposals to Ada language in the frame of Ada 9X revision process are also analyzed.

The specific problematic of the language implementation on top of the operating system is studied under the scope of the parallelism model. With this kind of implementation, program interactions with external environments can lead to problems, like the blocking of the corresponding operating system process, decreasing the program execution performance. A practical example of this kind of problems, the access to GKS (Graphic Kernel System) from Ada programs, is analyzed and the implemented solution is described.

INDICE

1 Introducción 1

2 Modelo jerárquico de paralelismo. Máquinas virtuales paralelas 9

2.1 Introducción 9

2.2 Niveles de paralelismo y concurrencia. Máquinas virtuales paralelas 11

2.3 Modelo de interacción entre Procesadores Virtuales 16

2.4 Granularidad del paralelismo 21

2.5 El problema de los bloqueos anidados 42

2.6 Sumario del capítulo 43

3 Niveles de paralelismo en un sistema. Modelo de paralelismo de Ada 47

3.1 Introducción 47

3.2 Niveles de paralelismo y concurrencia en un sistema convencional 48

3.3 Modelo de concurrencia definido por el lenguaje Ada 56

3.4 Conclusiones 64

4 Consideraciones sobre el modelo de paralelismo de Ada 65

4.1 Introducción 65

4.2 Modelo de interacción entre tareas de Ada 66

4.3 El anidamiento de las tareas y las reglas de ámbito 90

4.4 Terminación de las tareas Ada 101

4.5 La implementación de las llamadas condicionales y temporizadas 110

4.6 Sumario del capítulo 113

5 Implementación de Ada en diferentes entornos 117

5.1 Introducción 117

5.2 Partición y configuración 118

5.3 Implementación de Ada estándar 120

5.4 Implementación de Ada restringido o modificado. Ada 9X 124

5.5 Implementación de Ada sobre la máquina sistema operativo 138

5.6 Sumario del capítulo 142

6 Interfaz desde Ada al entorno externo 145

6.1 Introducción 145

6.2 Mecanismos que permiten el acceso al entorno externo 146

6.3 Los bloqueos anidados en Ada 151

6.4 Otros problemas relacionados 153

6.5 Sumario del capítulo 155

7 Acceso a GKS desde un programa concurrente Ada 157

7.1 Introducción 157

7.2 Problemas de acceso a GKS 158

7.3 Algunas posibles soluciones 162

7.4 Gestión de la entrada 165

7.5 Implementación de la solución resultante 167

7.6 Sumario del capítulo 170

8 Conclusiones 173

9 Bibliografía 179

1

Introducción

En la última década se ha producido una importante proliferación de computadores paralelos tanto en el campo experimental como comercial. Este desarrollo ha presentado una gran variedad de propuestas, desde arquitecturas más cercanas al modelo von Newmann, como los multiprocesadores (débil o fuertemente acoplados) y los procesadores matriciales (*array processors*), hasta alternativas más radicales, como las arquitecturas de flujo de datos y de reducción.

Sin embargo, a pesar de la mejor relación rendimiento/coste de estas máquinas frente a las máquinas secuenciales de altas prestaciones (p.ej. procesadores vectoriales), no se está produciendo el impacto esperado en el mundo de la computación.

El factor clave de esta situación es el escaso desarrollo de herramientas para programar adecuadamente este tipo de sistemas. La experiencia obtenida con las máquinas secuenciales demuestra que fue el desarrollo del software el que permitió el uso generalizado de los computadores convencionales.

Actualmente, la mayoría de las herramientas usadas para programar computadores paralelos están fuertemente ligadas a una arquitectura específica. Existe, por lo tanto, dependencia de las características del sistema, lo cual dificulta enormemente la transportabilidad del software, así como la labor del programador. Es necesario dise-

ñar lenguajes de programación que proporcionen independencia de la máquina pero que, además, puedan implementarse eficientemente en sistemas con diferentes arquitecturas.

Algunas de las alternativas que existen actualmente para programar computadores paralelos son las siguientes:

- Utilización de múltiples programas secuenciales que se comunican y sincronizan entre sí usando primitivas proporcionadas por una biblioteca en tiempo de ejecución. Existen dos alternativas típicas, utilizar múltiples copias de un programa que operan sobre diferentes datos (SCMD: *Single Code Multiple Data*), o bien utilizar programas diferentes (MCMD: *Multiple Code Multiple Data*). No se trata de una solución general sino eminentemente práctica. Presenta dependencia de la máquina, así como problemas debidos a que el compilador no puede tratar los múltiples programas como un único objeto.

- Utilización de lenguajes síncronos. En este tipo de lenguajes existe un único flujo de ejecución pero las operaciones se aplican a múltiples datos. Su uso está prácticamente restringuido a arquitecturas SIMD. Un ejemplo de este tipo es el C*.

- Utilización de lenguajes que no proporcionan una partición explícita del programa. Con estos lenguajes, el encargado de identificar el paralelismo en un programa es el compilador u otra herramienta automática. Dentro de esta clase, existen dos grandes grupos dependiendo de que el lenguaje esté libre de efectos laterales o no:

    - Lenguajes con efectos laterales. Con estos lenguajes se complica considerablemente el análisis autómatico del programa para identificar el paralelismo, no pudiéndose, generalmente, extraer todo el paralelismo. Los lenguajes secuenciales convencionales, como Pascal, o el lenguaje Lisp (no es puramente funcional) son ejemplos de esta clase.

  - Lenguajes sin efectos laterales. En los programas escritos con este tipo de lenguajes es sencillo identificar el paralelismo. En este grupo se incluyen diferentes tipos de lenguajes, como funcionales, de flujo de datos o lógicos.

- Utilización de lenguajes que proporcionan una partición explícita del programa. En estos lenguajes, el programador es el encargado de identificar el paralelismo del programa. Dentro de este grupo existe gran variedad de lenguajes, como el Ada, Multilisp o Linda (no es estrictamente un lenguaje, más bien se trata de un modelo de creación y sincronización de procesos mediante la abstracción del espacio de tuplas).

Las dos últimas alternativas, la paralelización automática de programas escritos con lenguajes sin efectos laterales y la utilización de lenguajes paralelos que permitan al programador expresar explícitamente el paralelismo del programa, se presentan como las dos soluciones más apropiadas.

Este trabajo se centra en esta última solución, el paralelismo explícito. El objetivo del mismo es evaluar si los lenguajes de este tipo pueden ser considerados como herramientas generales para programar máquinas paralelas. Este estudio se restringirá a máquinas con paralelismo asíncrono. En este conjunto, se va a incluir un amplio rango de máquinas, desde multiprocesadores con memoria compartida simétricos, hasta procesadores conectados por una red de prestaciones relativamente bajas.

Un lenguaje de programación permite cubrir el salto semántico que existe entre el modelo abstracto del problema que posee el programador y las características de la máquina física. Por lo tanto, un lenguaje debe proporcionar un modelo de programación adecuado a las necesidades del programador y, además, debe poder implementarse eficientemente sobre la máquina física correspondiente y, en general, sobre diversas máquinas para poder generar programas transportables.

El primer aspecto, la adecuación al programador de los lenguajes con paralelismo explícito, no se estudiará en este trabajo. A pesar de ello, vamos a incluir aquí un breve comentario sobre el tema. Existen numerosos autores que sostienen que este

tipo de lenguajes no definen un modelo adecuado, ya que obligan al programador a tener que controlar múltiples flujos de ejecución que se comunican, en vez de centrarse en las características del algoritmo. Según estos autores, los lenguajes de tipo implícito son más adecuados ya que liberan al programador de tener que ocuparse del paralelismo.

Existen, sin embargo, otras opiniones que están a favor de la utilización de lenguajes con paralelismo explícito ya que existen muchos algoritmos para los que la solución paralela es la más natural.

En nuestra opinión, no tiene demasiado sentido estudiar que alternativa proporciona un modelo más adecuado, de la misma forma, tampoco tiene sentido buscar una herramienta universal para programar computadores paralelos. Como ocurre con las máquinas secuenciales, a lo que se debe tender es a ofrecer diversos lenguajes que respondan a modelos muy diferentes pero que puedan implementarse eficientemente en una gran variedad de arquitecturas, y dejar que sea el programador el encargado de elegir un determinado lenguaje dependiendo de las características específicas del problema.

El trabajo va a ocuparse, por lo tanto, del segundo aspecto, esto es, analizar si los programas escritos con este tipo de lenguajes se pueden ejecutar eficientemente en entornos de muy diversas características sin necesidad de modificarlos. Algunos autores [SAR89] consideran que el paralelismo explícito no es adecuado para cumplir este objetivo ya que, cuando se ejecuta un programa de este tipo sobre diferentes máquinas, el rendimiento puede variar de manera radical dependiendo de la relación entre las necesidades de comunicación entre particiones del programa y la capacidad de comunicación proporcionada por la máquina. Puede ser necesario, por lo tanto, modificar el programa para que se ejecute eficientemente en una determina máquina. Con el paralelismo implícito, sin embargo, la partición automática se llevará a cabo teniendo en cuenta las características de la máquina por lo que no será necesario modificar el programa.

Este trabajo intenta rebatir estas opiniones y mostrar que los lenguajes con paralelismo explícito pueden ser una herramienta capaz de producir programas que pueden transportarse a sistemas con diferentes prestaciones.

El análisis del paralelismo explícito se realizará en el marco de la propuesta de un modelo abstracto de paralelismo. En este modelo un sistema está formado por una jerarquía de máquinas virtuales paralelas, donde cada máquina se define sobre la de nivel inferior.

Bajo el prisma de este modelo, la implementación de un lenguaje sobre diferentes máquinas paralelas se interpreta como la definición de la máquina virtual asociada al lenguaje (la máquina hipotética que ejecutaría los programas escritos en dicho lenguaje) sobre las máquinas físicas correspondientes.

La solución que se propondrá para resolver los problemas de transportabilidad de los programas con paralelismo explícito, es la utilización del **exceso de paralelismo**. Esta técnica consiste en utilizar mayor paralelismo en el programa que el existente en la máquina sobre la que se ejecuta para, de esta forma, esconder la latencia en las comunicaciones entre procesadores o entre un procesador y la memoria. Esta alternativa se ha estudiado en algunos trabajos sobre paralelismo relacionados con el modelo teórico P-RAM [VAL90] [SKI90].

En el estudio del paralelismo no se tiene en cuenta, en numerosas ocasiones, la influencia del sistema operativo centrándose en las características de la máquina física. Sin embargo, en la mayoría de los casos el lenguaje se implementa sobre el sistema operativo, y no directamente sobre el sistema físico. El modelo jerárquico de paralelismo va a permitir integrar el sistema operativo dentro del análisis de los lenguajes con paralelismo explícito, puesto que éste define una máquina virtual sobre la máquina física.

Cada lenguaje presenta características específicas que influyen en su implementación y que no pueden recogerse en el estudio genérico. Se ha considerado de interés, por lo tanto, particularizar dicho estudio a un determinado lenguaje con paralelis-

mo explícito. Se ha seleccionado Ada por su carácter de estándar y su gran difusión comercial, y por la existencia de numerosos trabajos sobre la utilización de este lenguaje para programar máquinas paralelas.

Además de aplicar a Ada las conclusiones generales sobre los lenguajes con paralelismo explícito, se examinarán características específicas que pueden influir en la implementación eficiente del lenguaje. Se estudiarán aspectos como la comunicación entre tareas mediante variables compartidas, el mecanismo de terminación de tareas y las implicaciones del mantenimiento de las reglas de ámbito.

Se analizará la implementación de Ada, tanto directamente sobre máquinas físicas con diferentes arquitecturas, como sobre sistemas operativos que sigan diferentes paradigmas (por ejemplo, sistemas operativos con procesos "ligeros").

Asimismo, se revisarán las propuestas de modificación del lenguaje en el campo de la programación de máquinas paralelas, principalmente la propuesta correspondiente al proceso de revisión Ada 9X [ADA9Xa] [ADA9Xb], evaluando las mejoras obtenidas respecto del Ada original (Ada 83).

Otro aspecto que se estudiará en este trabajo es la pérdida de rendimiento que puede producirse en la ejecución de programas con paralelismo explícito sobre máquinas con sistema operativo, debida a la ejecución de operaciones que bloqueen el correspondiente proceso del sistema operativo. En el caso de Ada, esta situación se produce, generalmente, debido a interacciones del programa con el entorno externo. Se analizará este tipo de problemas en el marco del modelo jerárquico de paralelismo, examinando sus repercusiones, que dependerán del tipo de implementación, y exponiendo algunas soluciones a los mismos.

En la parte final del trabajo, se planteará un ejemplo práctico de esta problemática: El acceso concurrente al estándar gráfico GKS. En este ejemplo se producen, además de bloqueos, problemas de reentrancia. Se describirá el diseño y la implementación de una solución para este caso práctico. Esta solución va a estar basada en

la utilización de un planificador que controle el acceso a GKS y elimine los problemas asociados a las operaciones bloqueantes.

Hay que resaltar que este trabajo se ocupa de uno de los aspectos importantes de la utilización de máquinas paralelas, la obtención del máximo rendimiento del sistema para lograr acelerar la ejecución de los programas. Sin embargo, el paralelismo permite otros logros importantes, y a veces contrapuestos con el rendimiento, como el aumento de la fiabilidad del sistema mediante el uso de técnicas de tolerancia a fallos, los cuales no se revisarán en este trabajo.

2

Modelo jerárquico de paralelismo. Máquinas virtuales paralelas

2.1 Introducción

Una de las principales dificultades que impide actualmente una mayor aceptación y utilización de los computadores paralelos reside en el desarrollo de software para este tipo de sistemas. Esta dificultad se debe a la dependencia del programador con respecto a las características de la máquina. Esta situación limita considerablemente la portabilidad de los programas paralelos.

Analizando la evolución de las máquinas secuenciales, se puede observar que uno de los factores claves para su utilización masiva fue el desarrollo de herramientas de programación independientes de la máquina subyacente: Los lenguajes de alto nivel. Un lenguaje de alto nivel define una máquina virtual que ejecuta programas escritos en dicho lenguaje. Esto facilita enormemente la portabilidad del software y la labor del programador que sólo necesita conocer las características de la máquina virtual. El programador únicamente seleccionará un lenguaje de alto nivel cuyas características sean adecuadas para expresar fácilmente la solución al problema que corresponda.

A pesar de la mayor complejidad y diversidad de los computadores paralelos, es necesario también en este tipo de máquinas desarrollar herramientas de programación independientes de la máquina que permitan un avance importante en el desarrollo de software paralelo. Siguiendo la analogía de las máquinas secuenciales, se puede considerar que estas herramientas definen máquinas virtuales paralelas.

En la primera parte de este capítulo se definirá un modelo jerárquico de máquinas virtuales paralelas asíncronas (nuestro estudio se centra en el paralelismo asíncrono). Este modelo abstracto permite definir la terminología necesaria y sirve de marco de referencia para el resto del trabajo.

Con la introducción de máquinas virtuales paralelas, el programador seleccionará aquella que sea más adecuada para resolver su problema. Pero, ¿puede olvidarse de las características de la máquina física como ocurría en las máquinas secuenciales?

Desgraciadamente, la respuesta es negativa. Dependiendo de las características de la máquina física y de la máquina virtual, es posible que esta última no pueda ser implementada eficientemente sobre el computador. Una máquina virtual con memoria compartida, por ejemplo, es difícil de implementar sobre una máquina con memoria distribuida.

Más allá, aunque una máquina virtual se pueda implementar con aceptable eficiencia, puede ser necesario que el programador modifique su programa para que éste se ejecute eficientemente debido a que los requisitos de comunicación del programa (granularidad) estén por encima de los soportados por el computador. Toda esta problemática será analizada en este capítulo.

En resumen, cuando se intenta programar una máquina paralela no solamente se restringe el tipo de máquina virtual que se puede utilizar al conjunto de máquinas que se puedan implementar de forma eficiente, sino que, una vez seleccionada la máquina virtual, el programa debe codificarse teniendo en cuenta las prestaciones del computador.

¿Existe alguna solución a esta situación? Nosotros recogemos una alternativa, la utilización de un exceso de paralelismo para esconder la latencia, que se presenta como una posibilidad prometedora. Se expondrán las características de este método analizando la manera en que proporciona al programador independencia de la máquina física.

Por último, se analizará una situación, a la que denominaremos el problema de los bloqueos anidados, en la que se producen bloqueos en cadena a lo largo de la jerarquía de máquinas virtuales disminuyendo el paralelismo en todo el sistema. En la parte final de este trabajo se analizarán casos prácticos de este problema.

2.2 Niveles de paralelismo y concurrencia. Máquinas virtuales paralelas

El paralelismo en un sistema se consigue, generalmente, mediante la utilización de varios procesadores que permiten la realización de múltiples actividades simultáneamente. En un sistema, sin embargo, se puede introducir concurrencia en otros niveles más allá del paralelismo hardware. El Sistema Operativo, por ejemplo, puede introducir concurrencia multiplexando procesos entre los distintos procesadores físicos existentes. De la misma forma, el entorno de ejecución de un lenguaje de programación concurrente puede introducir concurrencia al multiplexar las entidades concurrentes del lenguaje (por ejemplo tareas Ada) sobre procesos del Sistema Operativo.

El Sistema Operativo (S.O.) proporciona al usuario independencia de las características de la máquina física correspondiente. El usuario no tiene que conocer, por ejemplo, el número y tipo de procesadores que existen en el sistema o el modo en que están conectados los mismos. En cambio debe tener en cuenta el modelo de procesos definido por el S.O. y el tipo de mecanismo de comunicación que existe entre los mismos. El S.O., por lo tanto, define una máquina virtual paralela sobre la máquina física. De manera similar, el entorno de ejecución de un lenguaje concurrente proporciona al usuario independencia del S.O. y de la arquitectura subyacentes, definiendo una máquina virtual paralela cuyas características vienen dadas por las del lenguaje concurrente correspondiente.

Generalizando esta idea se puede describir un sistema como una jerarquía de máquinas virtuales paralelas, cada una de ellas definidas sobre la de nivel inferior. Según se va ascendiendo en la jerarquía aumenta la independencia de la máquina física, la cual corresponde al nivel inferior (fig. 2.1).

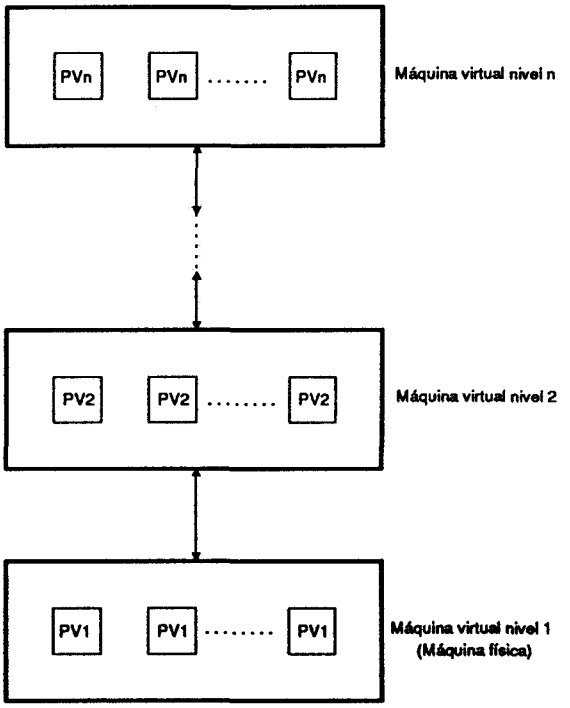

Fig. 2.1 Jerarquía de máquinas virtuales paralelas

Cada máquina virtual proporciona un conjunto de procesadores virtuales que se comunican entre sí utilizando los mecanismos de comunicación definidos por dicha máquina.

Este modelo, de alguna manera, sigue la filosofía del modelo definido por Tanembaum [TAN84] (descripción de un sistema como una jerarquía de máquinas virtuales), pero bajo el prisma del paralelismo.

En el modelo de Tanembaum el nivel N queda definido por un intérprete o traductor que se ejecuta en el nivel N-1. De forma análoga, en el modelo propuesto, la máquina virtual de nivel N queda definida por un **Entorno de Ejecución** que se

ejecuta en el nivel N-1 y se encarga, a partir de los servicios proporcionados por dicho nivel, de generar la funcionalidad correspondiente a la máquina de nivel N. Pueden definirse, por lo tanto, máquinas virtuales diferentes sobre una misma máquina o, al contrario, pueden definirse máquinas virtuales idénticas sobre diferentes máquinas. Por ejemplo, sistemas con la misma arquitectura con diferentes Sistemas Operativos (el S.O. es el Entorno de Ejecución que define la máquina virtual "Sistema Operativo") o sistemas con diferentes arquitecturas con la misma máquina "Sistema Operativo".

Una de las funciones principales del Entorno de Ejecución es la multiplexación de los procesadores de la máquina que define sobre los procesadores de la máquina de nivel inferior. La asignación entre los procesadores de ambos niveles puede ser estática o dinámica.

En la asignación estática, cada procesador del nivel superior está asociado a un determinado procesador del nivel inferior durante toda su existencia (fig 2.2).

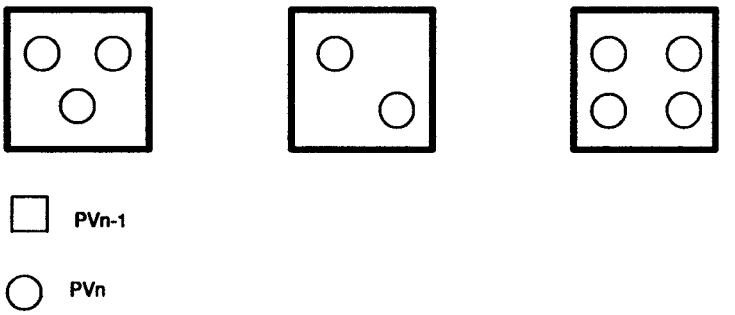

Fig. 2.2 Asignación estática

Una asignación dinámica implica que un determinado procesador del nivel superior, a lo largo de su existencia, puede estar asociado a diferentes procesadores del nivel inferior. En la figura 2.3 se puede ver un ejemplo de asignación dinámica en el cual cada procesador del nivel inferior elige para ejecutar, a lo largo del tiempo, diferentes procesadores de nivel superior. Este tipo de asignación proporciona dinámicamente un reparto de la carga entre los procesadores de nivel inferior.

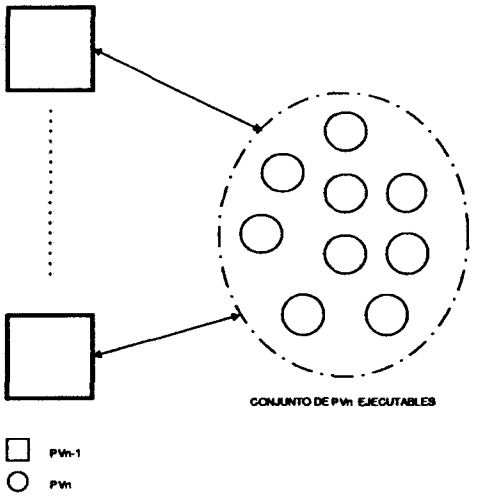

Fig.2.3 Asignación dinámica

Para realizar la asignación entre los procesadores de ambos niveles, el Entorno de Ejecución debe tener en cuenta el estado de los mismos. Un procesador puede estar en dos estados: Ejecutable o bloqueado. El estado de bloqueo se produce, en general, cuando un procesador debe esperar por un determinado evento para poder continuar su ejecución. Cuando un procesador se bloquea, el procesador de nivel inferior que tiene asignado en ese instante puede asociarse a otro procesador que esté en estado ejecutable para, de esta forma, aprovechar al máximo el paralelismo de la máquina de nivel inferior. En el caso de asignación dinámica, cuando un procesador se bloquea, el procesador de nivel inferior correspondiente selecciona un procesador del conjunto total de procesadores ejecutables. En el caso de asignación estática, el procesador de nivel inferior buscará un procesador ejecutable entre los que están estáticamente asociados a él.

Si el número de procesadores del nivel superior es mayor que el número de procesadores de nivel inferior puede darse el caso de que un determinado procesador en estado ejecutable no pueda ser ejecutado en ningún procesador de nivel inferior, en un cierto instante, al no haber ninguno disponible. En este caso, y de acuerdo a una determinada política de planificación, el Entorno de Ejecución debe elegir, en

cada momento, entre los procesadores en estado ejecutable, el subconjunto que será ejecutado.

El Entorno de Ejecución multiplexa **m** procesadores de nivel N sobre **n** procesadores de nivel N-1. Dependiendo de los valores de **m** y **n** pueden darse los siguientes casos:

- **m=n** (en general **m≤n**). Todos los procesadores en estado ejecutable pueden ser ejecutados. En algunos casos, como se analizará más adelante, esta asignación puede llevar a una utilización poco eficiente de la máquina de nivel inferior.

- **m>n (n≠1)**. Se trata del caso más general. Black [BLA90] lo denomina modelo multirrutina: Una generalización del concepto de corrutina para varios procesadores. Existirán, en ciertos momentos, algunos procesadores de nivel N en estado ejecutable que no pueden ser ejecutados. Esta situación en la cual la máquina de nivel superior posee mayor paralelismo que la de nivel inferior, suele denominarse *parallel slackness* [VAL90] [SKI90]. Como se verá más adelante, esta característica puede ser necesaria en algunos casos para la utilización eficiente de la máquina de nivel inferior.

- **m>n (n=1)**. Corresponde a un modelo de corrutina (un único procesador de nivel N-1) en la definición de Black. Con esta correspondencia en cada momento sólo se ejecuta un único procesador de nivel N.

El tipo de multiplexación que utiliza el Entorno de Ejecución para definir una nueva máquina sobre otra ya existente va a depender de las características de ambas máquinas. A continuación se van a analizar tres factores que influyen en la selección del tipo de multiplexación, a saber, el modelo de comunicación entre procesadores que existe en cada máquina, la granularidad del paralelismo de cada máquina y los bloqueos anidados.

2.3 Modelo de interacción entre Procesadores Virtuales

Los modelos de interacción entre los procesadores de un determinado nivel están fuertemente influenciados por la utilización o no de memoria compartida para comunicarse entre sí.

- Sin memoria compartida (memoria distribuida). Debe existir algún mecanismo de paso de mensajes, o equivalente, que permita a los procesadores interaccionar entre sí (fig.2.4).

- Memoria compartida. Los procesadores pueden interaccionar usando la memoria compartida (fig.2.5). Cada procesador puede tener también su propia memoria privada.

- Pueden existir máquinas que sigan un modelo mixto (caso más general) en el que no todos los procesadores comparten memoria sino que existen subconjuntos de forma que, dentro de cada subconjunto, los procesadores interaccionan mediante memoria compartida. La comunicación entre procesadores pertenecientes a diferentes subconjuntos se lleva a cabo mediante algún mecanismo de paso de mensajes o equivalente (fig.2.6).

Se pueden distinguir dos tipos de interacciones entre los componentes de una máquina paralela: Comunicación y sincronización. La comunicación consiste en el paso de datos entre componentes. La sincronización implica la interacción entre los procesadores para satisfacer una serie de restricciones en el orden en el que deben ocurrir ciertas acciones.

Estos conceptos están bastante relacionados, ya que la comunicación implica muchas veces sincronización y la sincronización puede interpretarse como una comunicación sin datos. Debido a esto, muchas veces se da un tratamiento uniforme a ambas operaciones (la cita del lenguaje Ada, por ejemplo, es un mecanismo de sincronización y comunicación simultáneamente).

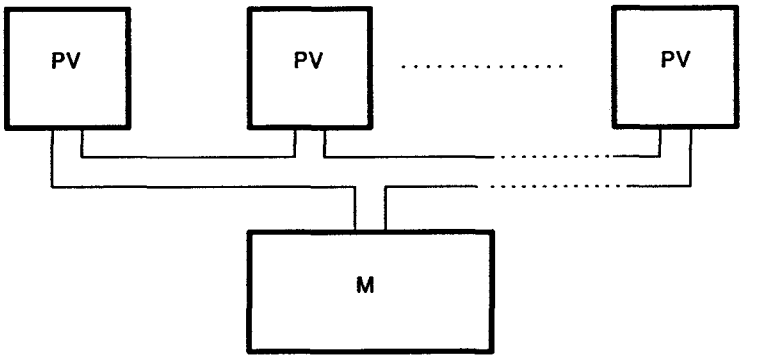

Fig. 2.4 Máquina virtual con memoria compartida

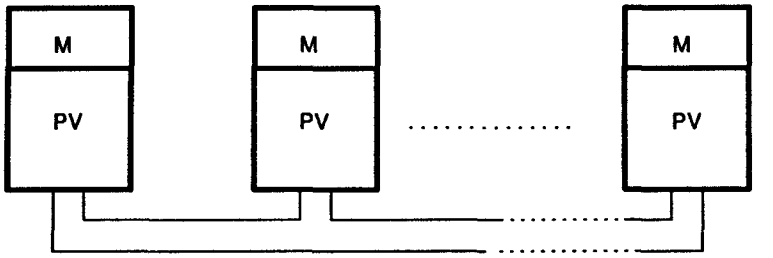

Fig. 2.5 Máquina virtual sin memoria compartida

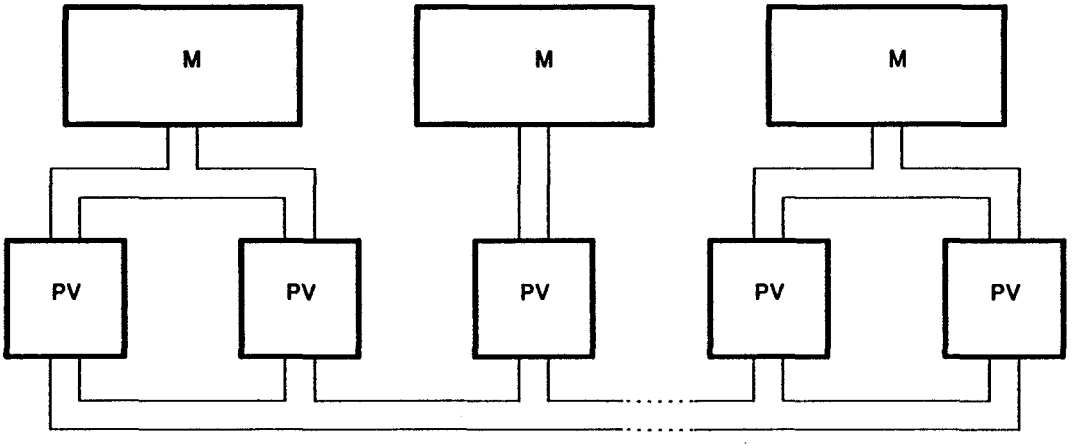

Fig. 2.6 Máquina virtual con un modelo mixto

Sin embargo, es importante distinguir, para los análisis que posteriormente se realizarán, dos tipos de interacciones entre los componentes de una máquina paralela: Interacción con sincronización y sin sincronización.

Una interacción con sincronización, además de una posible comunicación de datos, requiere una sincronización entre dos o más procesadores de la máquina. No es posible, generalmente, prever la duración de dicha interacción ya que depende del estado de los procesadores que deben sincronizarse, lo cual es difícil de predecir en una máquina asíncrona. Cuando un procesador inicia una interacción de este tipo, generalmente, pasa a un estado de bloqueado y el procesador de nivel inferior asociado al mismo ejecuta otro procesador de nivel superior que no esté bloqueado, si existe alguno. Un ejemplo de este tipo puede ser un mecanismo de paso de mensajes síncrono en una máquina sin memoria compartida. Con este mecanismo, el procesador que manda el mensaje se queda bloqueado hasta que el otro procesador lo reciba. En máquinas virtuales con memoria compartida, un posible ejemplo sería el modelo de memoria del lenguaje Linda [CAR89a][AHU86] o las "Estructuras-I" de Arvind [NIK89], en ambos casos si un procesador intenta leer un dato que no esté presente en la memoria se queda bloqueado hasta que otro procesador escriba dicho dato en la misma.

Una interacción sin sincronización, por otro lado, implica únicamente una comunicación de datos. El tiempo que tarda en realizarse la interacción (se le suele llamar **latencia** de la comunicación) es predecible, dentro de unos limites, ya que no es necesario que ningún otro procesador esté en un estado determinado para llevarse a cabo la transferencia. Normalmente, mientras dura la comunicación no se produce un cambio de contexto aunque, como analizaremos más adelante, puede ser una posibilidad interesante en algunos casos. Un ejemplo de este tipo de interacción en máquinas sin memoria compartida puede ser un mecanismo de paso de mensajes asíncrono, en el cual el procesador que envía el mensaje no espera que sea recibido para continuar su ejecución. En máquinas con memoria compartida, un ejemplo sería el acceso de un procesador a la memoria compartida.

2.3.1 Definición de máquinas virtuales con distintos modelos de interacción

La definición de máquinas que sigan un determinado modelo de interacción sobre máquinas virtuales que respondan a otro modelo puede ser problemática. Analicemos los diferentes casos:

- Si la máquina de nivel inferior proporciona memoria compartida no existen problemas a la hora de implementar máquinas que sigan diferentes modelos ya que serán más restrictivas que la máquina de nivel inferior. El mecanismo de comunicación existente en el nivel superior puede ser simulado eficientemente usando la memoria compartida del nivel inferior.

- Si la máquina de nivel inferior no proporciona memoria compartida para todos los procesadores (modelo sin memoria compartida o modelo mixto) pueden existir problemas para implementar sobre ella una máquina en la que algunos procesadores utilicen memoria compartida para comunicarse. Es necesario que el Entorno de Ejecución implemente la memoria compartida usando el mecanismo de comunicación del nivel inferior. En la mayoría de los casos esto puede resultar difícil y/o poco eficiente de implementar. Va a depender de factores como la frecuencia de los accesos a la memoria compartida y la localidad de los mismos, así como de las características del mecanismo de comunicación del nivel inferior. Ante estas dificultades, y volviendo a nuestro modelo, se pueden buscar alternativas restringiendo el uso de la máquina de nivel inferior:

  - En el caso de que se trate de una máquina con un modelo mixto, los procesadores de nivel superior que compartan memoria se asignarán a procesadores de nivel inferior que pertenezcan a un mismo subconjunto con memoria compartida. Si todos los procesadores de nivel superior comparten memoria (máquina con memoria compartida), toda la máquina de nivel superior se asocia a un único subconjunto de la máquina inferior. Se trata, por lo tanto, de una correspondencia de tipo **m>n**, siendo **n** el número de procesadores del subconjunto de la máquina inferior utilizado.

- Si la máquina inferior sigue un modelo sin memoria compartida, los procesadores de nivel superior que compartan memoria se asignarán a un único procesador de nivel inferior utilizando la memoria privada del mismo como memoria compartida de los procesadores de nivel superior (correspondencia **m>n**). En el caso de que todos los procesadores de la máquina de nivel superior compartan memoria (máquina con memoria compartida), se asignarán a un único procesador de nivel inferior (correspondencia **m>n, n=1**).

Estas restricciones, aunque facilitan la implementación de máquinas con memoria compartida sobre máquinas con memoria distribuida (o con modelo mixto), implican la no utilización de todo el paralelismo que proporciona la máquina de nivel inferior y, por lo tanto, no proporcionan una solución general.

En el siguiente apartado, bajo el concepto de granularidad, se generalizará este análisis y se expondrá en qué condiciones se puede aprovechar todo el paralelismo de la máquina inferior con independencia de los modelos de interacción existentes en ambas máquinas.

Esta generalización se basa en romper la clásica dicotomía entre memoria compartida y memoria distribuida, y distinguir los mecanismos de interacción entre procesadores por sus prestaciones. Es evidente que, por ejemplo, no es lo mismo una máquina con memoria distribuida en la que sus procesadores se comunican mediante un enlace de transputer (hasta 20 Mbits/s) que otra en la que se comunican mediante conexiones RS232 a 300 baudios. Así, el problema de construir una máquina que utiliza memoria compartida sobre una máquina con memoria distribuida (o con modelo mixto), se generaliza al de construir una máquina que necesita un mecanismo de interacción de mayores prestaciones que las proporcionadas por el mecanismo de interacción de la máquina de nivel inferior.

En las conclusiones del artículo [CME89], por ejemplo, se comentan las dificultades de implementar Concurrent C (máquina virtual con memoria distribuida) sobre un sistema distribuido basado en Ethernet y UNIX debido a las diferentes características de los mecanismos de comunicación de las máquinas virtuales definidas

por Concurrent C y UNIX. Los procesos de Concurrent C (procesadores de nivel superior) tienden a intercambiar mensajes demasiado frecuentemente con relación a las prestaciones proporcionadas por la red Ethernet.

Es importante resaltar que esta visión unificadora entre la memoria compartida y la memoria distribuida está referida a la comunicación de datos entre los procesadores, pero no se puede aplicar a la lectura de las instrucciones por parte de un procesador. Las instrucciones que ejecuta un procesador las debe obtener de una memoria directamente accesible por él (ya sea local o compartida) para obtener una solución eficiente. Las técnicas que se verán en el siguiente apartado para el caso de acceso a datos (aumentar la granularidad y esconder latencia) no son fácilmente aplicables al acceso a instrucciones.

Debido a esto, es más difícil realizar eficientemente asignación dinámica (un procesador de nivel N puede ser ejecutado por diferentes procesadores de nivel N-1 a lo largo de su existencia) en máquinas con memoria distribuida (o modelo mixto), ya que será necesario realizar una migración del código asociado al procesador de nivel superior a una memoria accesible por el nuevo procesador. En máquinas con memoria compartida, en cambio, esta migración no será necesaria si el código está almacenado en la memoria accesible por los procesadores de nivel inferior, por lo tanto, puede llevarse a cabo de una forma eficiente una asignación de carácter dinámico.

2.4 Granularidad del paralelismo

La granularidad de una máquina paralela expresa la relación entre la capacidad de comunicación entre los procesadores y la capacidad de procesamiento que proporciona la misma.

En una máquina paralela es importante que exista un buen equilibrio entre la capacidad de procesamiento y la comunicación para que no se creen cuellos de botella que impidan la utilización eficiente de la máquina. Generalmente, el cuello de botella de una máquina paralela se encuentra en la comunicación, ya que es más

difícil aumentar las prestaciones de la red de comunicación que la capacidad de procesamiento.

La granularidad, por lo tanto, está fuertemente influenciada por la latencia en las comunicaciones, ya sea entre procesadores y memoria, o entre procesadores.

Generalmente, se aplica el término de máquina con grano fino a aquella con un buen equilibrio entre el procesamiento y la comunicación, y máquina con grano grueso a aquella en la que no existe un buen equilibrio. La granularidad de una máquina con memoria compartida es generalmente más fina que de la de una con memoria distribuida.

El concepto de granularidad se aplica también a los programas paralelos. En este caso, refleja la necesidades de comunicación entre los procesos del programa (nombre genérico de las entidades paralelas del programa) con respecto a las necesidades de procesamiento del mismo. La granularidad de un programa podría definirse como el número medio de instrucciones de un proceso que se ejecutan por cada comunicación o, dicho de otra manera, el tamaño medio de los intervalos de ejecución de instrucciones entre dos comunicaciones.

Así, un programa tendrá un paralelismo de grano fino si los procesos del mismo se comunican con mucha frecuencia (cada pocas instrucciones ejecutadas) y, por el contrario, tendrá grano grueso si se ejecutan muchas instrucciones entre dos comunicaciones.

La ejecución de un programa paralelo sobre una máquina cuya granularidad sea mayor que la necesitada por el programa llevará a una utilización poco eficiente de la máquina.

Sea **s** la latencia media en la comunicación de una determinada máquina e **i** el número medio de instrucciones ejecutadas por un procesador durante un segundo. Sea **j** el número medio de instrucciones que se ejecutan por cada comunicación en un determinado programa. La utilización de cada procesador cuando la máquina ejecuta dicho programa será $U = c/(c+s)$ siendo $c = j/i$ el tiempo medio entre comunicaciones.

Esto es, en cada intervalo de c+s unidades de tiempo sólo se realiza procesamiento útil durante c unidades de tiempo. Durante las restantes s unidades el procesador está bloqueado mientras dura la comunicación, ya sea con la memoria compartida o con otro procesador.

Si c es aproximadamente igual a s, la utilización de cada procesador estará alrededor del 50%. Con c cuatro veces mayor que s la utilización llegará al 80%.

Para aumentar la utilización de los procesadores de una máquina por parte de un determinado programa, existen dos alternativas: Aumentar el tiempo medio entre comunicaciones c, o bien disminuir la latencia en la comunicación s.

La latencia en la comunicación puede disminuirse, además de por mejoras tecnológicas y arquitectónicas, "acercando" los datos al procesador correspondiente mediante la utilización de una jerarquía de memoria como en el sistema Paradigm [CHE91]. También se puede disminuir utilizando sólo un subconjunto de los procesadores de la máquina, como vimos en el apartado anterior, de forma que la comunicación entre dichos procesadores sea más rápida.

En cuanto al tiempo entre comunicaciones, se puede aumentar reestructurando el programa de forma que el número de instrucciones que se ejecutan por cada comunicación j sea mayor, esto es, aumentando la granularidad del programa (otra alternativa sería disminuir la potencia i de los procesadores).

Así, cuando un programador desea resolver un problema usando una determinada máquina, debe tener en cuenta, para poder realizar un programa que se ejecute eficientemente, tanto las características específicas del problema como las de la máquina.

Esta situación dificulta considerablemente la transportabilidad del programa, así como la labor del programador que no puede centrarse únicamente en las características del problema, como ocurre cuando programa máquinas secuenciales, y debe tener en cuenta las características de la máquina que se va a utilizar.

En [CAR89b], Carriero y Gelernter exponen un interesante análisis sobre la reestructuración de programas paralelos para adaptarlos a diferentes máquinas. Su exposición se basa en distinguir tres clases conceptuales de paralelismo y tres paradigmas de programación paralela.

Las clases conceptuales son: Paralelismo de resultados (todos los componentes del resultado son calculados en paralelo), paralelismo de agenda (existe una lista de actividades que hay que realizar para resolver el problema y un conjunto de "trabajadores" idénticos que las llevarán a cabo) y paralelismo de especialistas (un conjunto de "trabajadores" especializados cooperan para resolver el problema).

Los paradigmas de programación son: Estructuras de datos "vivas" (cada proceso se crea, realiza el procesamiento sobre el dato asociado y termina dejando un resultado), paso de mensajes (procesos que se comunican explícitamente mediante mensajes) y estructuras de datos distribuidas (todos los procesos acceden a dichos datos).

El proceso de escribir un programa paralelo contaría de las siguientes fases:

- Elegir la clase conceptual que se adapte mejor al problema.
- Escribir el programa usando el paradigma más natural para el modelo conceptual elegido, independientemente de la máquina a la que esté destinado.
- Si el programa resultante no ejecuta eficientemente en la máquina, transformarlo cambiando el paradigma utilizado hasta obtener una solución eficiente.

Esta situación, en la que el programador debe estructurar su programa dependiendo de las características de la máquina que lo va a ejecutar, es un escollo importante para el desarrollo de software paralelo. A continuación, vamos a analizar una alternativa, la **multiplexación** que proporciona al programador mayor independencia de la máquina.

Antes, sin embargo, vamos a volver a nuestro modelo de máquinas virtuales paralelas para aplicar al mismo el concepto de granularidad.

Cuando se define una máquina virtual sobre otra ya existente, se debe tener en cuenta la granularidad proporcionada por la máquina inferior frente a la granularidad necesitada por la máquina virtual que se define. ¿Como se puede evaluar la granularidad que necesita una determinada máquina?

La granularidad que necesita una máquina virtual estará caracterizada por la granularidad de los programas que ejecuta. La granularidad de una máquina Ada, por ejemplo, vendrá determinada por la granularidad típica de los programas Ada.

Aunque este concepto no sea absoluto, ya que para una determinada máquina existirán programas de muy diversa granularidad, sin embargo puede servir para comparar necesidades de diferentes máquinas. Así, por ejemplo, se puede escribir un programa en Occam con mayor granularidad que otro programa escrito en Concurrent C. Sin embargo, en general, los programas en Occam, por las características del lenguaje, tienen un grano más fino que los de Concurrent C y, por lo tanto, la máquina Occam necesita una granularidad más fina que la máquina Concurrent C.

Multiplexación y exceso de paralelismo

El factor clave que dificulta la utilización eficiente de las máquinas paralelas es, como se expuso anteriormente, la latencia en la comunicación, ya que durante el tiempo que dura la transferencia el procesador no realiza trabajo útil.

Una técnica para esconder esta latencia es la multiplexación de los procesos del programa (en general, los procesos de la máquina de nivel superior) sobre los procesadores de la máquina inferior, de forma que cuando un proceso inicia una comunicación (por ejemplo, una referencia a la memoria compartida), el procesador asociado al mismo realiza un cambio de contexto y pasa a ejecutar otro proceso. Esta técnica, como se expuso antes, se utiliza normalmente para las interacciones que implican sincronización (p.ej. un mecanismo de paso de mensajes síncrono) ya que no se puede predecir su duración.

La propuesta es, por lo tanto, utilizar la misma técnica para ambos tipos de interacción, con o sin sincronización, de forma que siempre que un proceso inicie una interacción, ya sea el acceso a la memoria compartida o mandar un mensaje a otro proceso de forma síncrona, se produzca un cambio de contexto y se pase a ejecutar otro proceso. Así, existirá un tratamiento uniforme de ambos tipos de interacción, a pesar de sus diferentes características.

Una posible analogía puede encontrarse en los sistemas operativos con multiprogramación. El acceso de un proceso a datos almacenados en un disco, por ejemplo, tiene una duración predecible dentro de unos límites. Por otra parte, una operación de sincronización entre procesos no puede predecirse cuando terminará. Sin embargo, el sistema operativo trata a ambas operaciones de la misma forma, se realiza un cambio de contexto y se pasa a ejecutar otro proceso.

Con esta técnica y con un número suficientemente mayor de procesos (procesadores de nivel superior) que de procesadores de nivel inferior, o sea, una correspondencia **m>n** con **m** suficientemente mayor, se puede lograr una utilización eficiente de la máquina aunque la granularidad de ésta no sea adecuada.

¿Cuanto más paralelismo debe tener la máquina de nivel superior que la máquina inferior para poder utilizarla efectivamente? ¿Qué relación debe existir entre **m** y **n**?

Se han realizado estudios de este tipo en el entorno del modelo teórico P-RAM. En primer lugar, se revisarán brevemente las conclusiones de dichos estudios.

2.4.1 Modelo P-RAM

El modelo **P-RAM** (*Parallel Random Access Machine*) es un modelo teórico de computación paralela frecuentemente utilizado para la evaluación de la complejidad de algoritmos paralelos y para realizar estudios generales sobre el paralelismo. Una P-RAM consiste en un conjunto de procesadores con memoria compartida, cada uno de los cuales puede realizar en una unidad de tiempo un acceso a la memoria

local del procesador, un acceso a la memoria global y la ejecución de una instrucción.

De esta definición se desprende que el modelo P-RAM se apoya en algunas suposiciones que son difícilmente realizables en la práctica. En primer lugar, en esta máquina existe una comunicación con la memoria compartida por cada instrucción. Esta granularidad es demasiado fina para poder ser implementada eficientemente.

Por otro lado, el tiempo de acceso a la memoria compartida en este modelo es constante e independiente del número de procesadores. Las implementaciones reales, debido a la latencia en la red de interconexión entre los procesadores y la memoria, suelen tener un tiempo de acceso medio del orden de **log p**, siendo **p** el número de procesadores. Además, hay que tener en cuenta que el tiempo de acceso a la memoria depende también de la tasa de accesos a la misma. Así, si la tasa de accesos a memoria compartida excede el ancho de banda de la misma, entonces la latencia será mayor debido a los tiempos de espera necesarios.

Estas discrepancias entre el modelo teórico y las máquinas existentes implican que un determinado programa (máquina virtual), que se ejecutaría eficientemente sobre una P-RAM, puede no ejecutarse de manera eficiente sobre una máquina real. En [SKI90], se analiza este problema comparando el modelo P-RAM con diferentes arquitecturas paralelas clásicas. Una de las conclusiones del artículo es que, tanto en arquitecturas MIMD fuertemente acopladas como en MIMD débilmente acopladas conectadas en hipercubo, se puede ejecutar eficientemente un programa (máquina virtual) si se utiliza la técnica de multiplexación expuesta anteriormente para esconder la latencia. Así, un programa que se ejecuta sobre una P-RAM utilizando **p** procesos debe utilizar **p/log p** procesadores en ambas arquitecturas, de forma que cada procesador ejecuta **log p** procesos para esconder la latencia en el acceso a la memoria compartida y en la transferencia de mensajes por el hipercubo respectivamente.

Esto implica que un algoritmo con 10.000 procesos concurrentes podrá ejecutar eficientemente sobre máquinas que tengan del orden de 1.000 procesadores.

La correspondencia entre procesadores será, por lo tanto, del tipo (**m>n** con **m= n•logn**) siendo **m** el número de procesadores de la máquina superior y **n** de la inferior.

Esta diferencia de paralelismo (*parallel slackness*) y la aplicación de la técnica de multiplexación permiten "acercar" las características de las máquinas reales a las características teóricas de la P-RAM. De esta forma, la latencia no constante de las máquinas reales (la latencia crece del orden de **log p**) queda escondida utilizando la cantidad suficiente de multiplexación (**log p** procesos por procesador).

Existen otros modelos teóricos de paralelismo como el **bulk-synchronous parallel** (BSP) o el XPRAM [VAL90]. Estos modelos son menos exigentes que el modelo P-RAM estando, por lo tanto, más próximos a las máquinas reales. En estos modelos se relaja la necesidad de sincronización por cada instrucción existente en el modelo P-RAM, realizándose las sincronizaciones cada cierto número de unidades de tiempo. El BSP y el XPRAM son máquinas con mayor granularidad y, por lo tanto, más fáciles de implementar. En los estudios realizados sobre estos modelos se concluye también con la necesidad *parallel slackness* y multiplexación para la utilización eficiente de la máquina.

Vamos a realizar, a continuación, un análisis simplificado que nos permitirá obtener algunos resultados, entre otros, la relación entre **m** y **n** para lograr una utilización eficiente de una determinada máquina.

2.4.2 Análisis cuantitativo de la multiplexación

Supongamos una determina máquina paralela con **n** procesadores, una latencia en la comunicación **s**, una tasa media de **i** instrucciones ejecutadas por cada procesador en un segundo y un *overhead* de **k** unidades de tiempo. En **k** se incluyen el tiempo durante el cual el procesador no está realizando trabajo útil sino otras operaciones como iniciar la comunicación (una vez iniciada se pasa a ejecutar otro proceso), realizar un cambio de contexto o crear un nuevo proceso.

Supóngase, por otro lado, un programa paralelo (en general, una máquina virtual paralela) con **m** procesos concurrentes (**m** procesadores virtuales de nivel superior). Cada proceso ejecuta, en término medio, **j** instrucciones por cada comunicación.

Cuando un proceso inicia una transferencia, se realiza un cambio de contexto y el procesador asociado pasa a ejecutar instrucciones de otro proceso. Este nuevo proceso, a su vez, dejará de ser ejecutado cuando inicie una transferencia, y así sucesivamente.

Para esconder totalmente la latencia de la comunicación es necesario que exista el suficiente número de procesos para ocupar dicha latencia. Cada proceso aporta **c** unidades de cómputo, siendo **c** el tiempo medio entre comunicaciones **c= j/i**, más **k** de *overhead* debido al inicio de la comunicación y al cambio de contexto. Si **t** **(t=m/n)** es el número de procesos asociados a cada procesador, para ocupar la latencia de una transferencia tendremos **(t-1)•(c+k)** unidades de tiempo proporcionadas por todos los procesos exceptuando el que inició la transferencia. Sin embargo, sólo necesitamos cubrir **(s-k)** unidades, ya que es necesario descontar el cambio de contexto correspondiente al proceso que inicia la transferencia. Por lo tanto, para esconder totalmente la latencia se debe cumplir:

$(t-1)\bullet(c+k) \geq (s-k)$; o bien **$t\bullet k+(t-1)\bullet c \geq s$**

(Realmente, se debería haber distinguido dentro de **k** entre el tiempo de inicio de la transferencia y el tiempo de cambio de contexto y, únicamente, haber descontado de **s** el tiempo de cambio de contexto. Sin embargo, hemos descontado el término **k** completo ya que de esta forma se simplifican los cálculos al existir una única variable que recoja el *overhead.* Por otro lado, esta simplificación sería problemática únicamente en el caso límite en el que descontando **k** completo, la latencia pareciese totalmente cubierta, mientras que descontando sólo el cambio de contexto no estuviese totalmente cubierta. Aun así, el error al calcular la utilización del procesador sería bastante pequeño).

La utilización de cada procesador quedaría de la siguiente forma (figura 2.7):

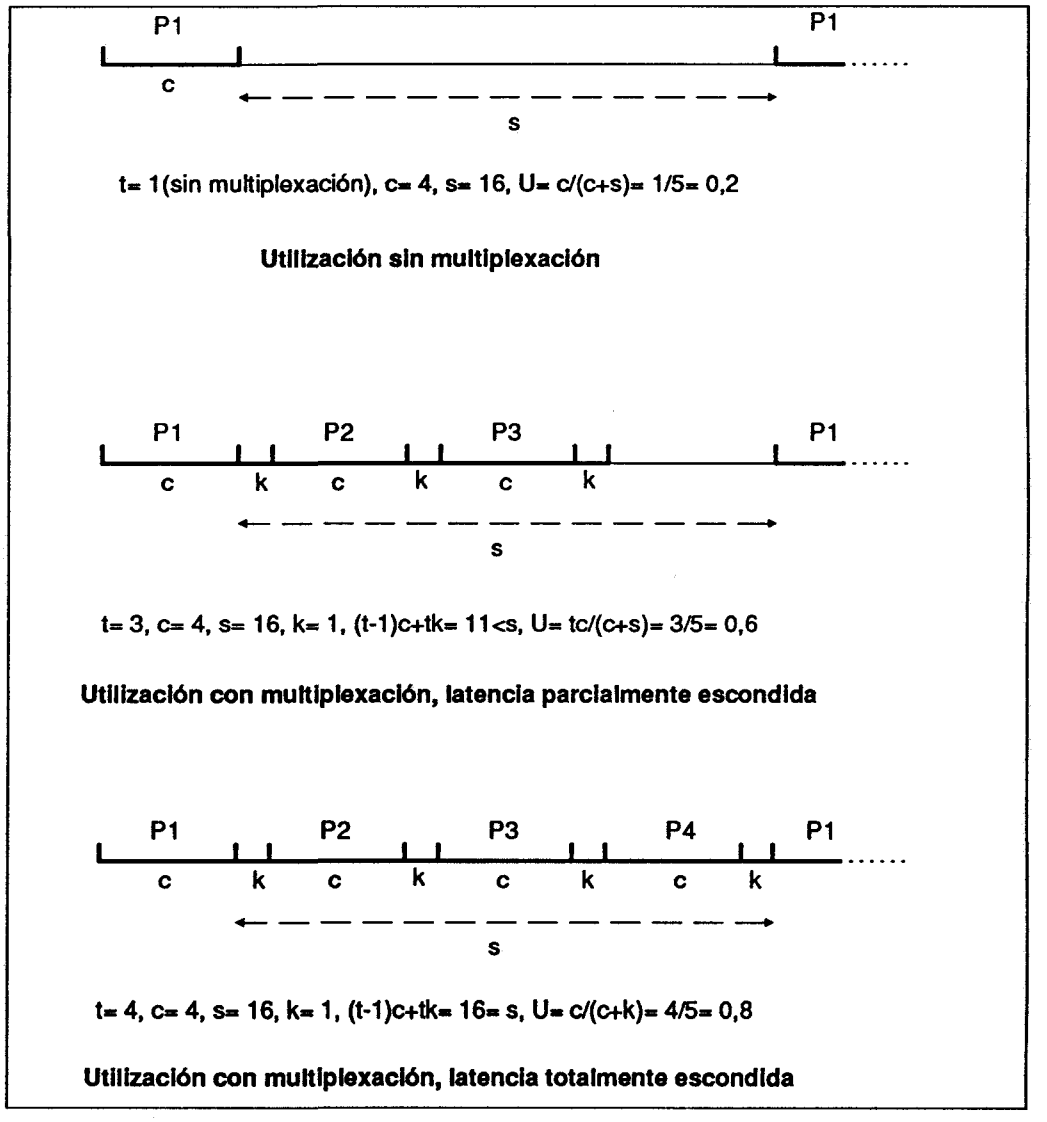

Fig.2.7 Utilización del procesador en diferentes casos

Si $t \cdot k+(t-1) \cdot c \geq s$ (latencia totalmente escondida): $U= c/(c+k)$

Si $t \cdot k+(t-1) \cdot c<s$: $U= t \cdot c/(c+s)$; para $t=1$ (no existe multiplexación): $U= c/(c+s)$

Cuando está totalmente escondida la latencia, la utilización de cada procesador es independiente de la misma. Depende únicamente de la granularidad del programa respecto al *overhead* de la máquina.

Debido a que, normalmente, **s** es bastante mayor que **k**, el aumento del número de procesos asociados a cada procesador desde la no multiplexación (**t=1**) hasta que existan los suficientes para esconder la latencia, implica un aumento en la utilización de cada procesador desde **c/(c+s)** hasta **c/(c+k)** (figura 2.8).

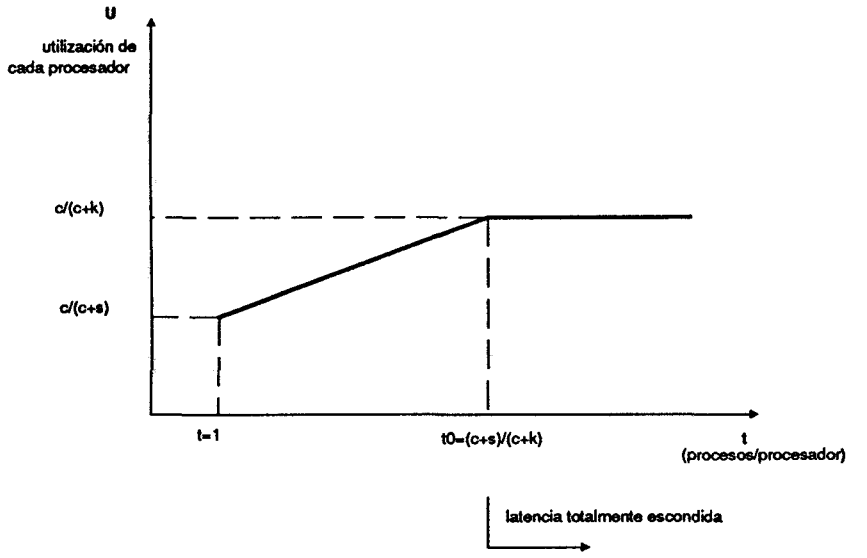

Fig. 2.8 Utilización en función de procesos/procesador

El número de procesos por procesador mínimo para ocultar la latencia será $t_0=(c+s)/(c+k)$ y, por lo tanto, el número total de procesos mínimo para ocultar la latencia será $m_0= n\cdot(c+s)/(c+k)$.

Para **t** mayor que t_0, la utilización del procesador permanece constante.

Esta situación indica que sobra paralelismo en el programa y, por lo tanto, se puede ejecutar en una máquina con más procesadores.

Si **k** y **s** son de un orden de magnitud similar, la mejora obtenida al utilizar la multiplexación puede ser muy pequeña o, incluso, nula. Teniendo en cuenta que la introducción de multiplexación complica el diseño de la máquina, como veremos más adelante, puede carecer de interés introducir multiplexación en estos casos.

Speedup

Otro parámetro que se puede estudiar es el *speedup*. Este parámetro se define como la razón entre el tiempo que tarda en ejecutarse un programa en una máquina secuencial respecto al tiempo que tarda en una máquina con **n** procesadores. Este valor refleja el aprovechamiento de los procesadores de la máquina. Así, idealmente, el *speedup* de una máquina con **n** procesadores debería ser **n**.

speedup= (tiempo en un procesador)/(tiempo en **n** procesadores)= **n**•**U**

En la figura 2.9, se puede observar la variación de este parámetro en función del número de procesadores que varía entre 2 (mínima máquina paralela) y **m** (número de procesos concurrentes). Hasta un cierto umbral $\mathbf{n_0}$= m•(c+k)/(c+s) el *speedup* crece linealmente con el número de procesadores, ya que la latencia queda totalmente

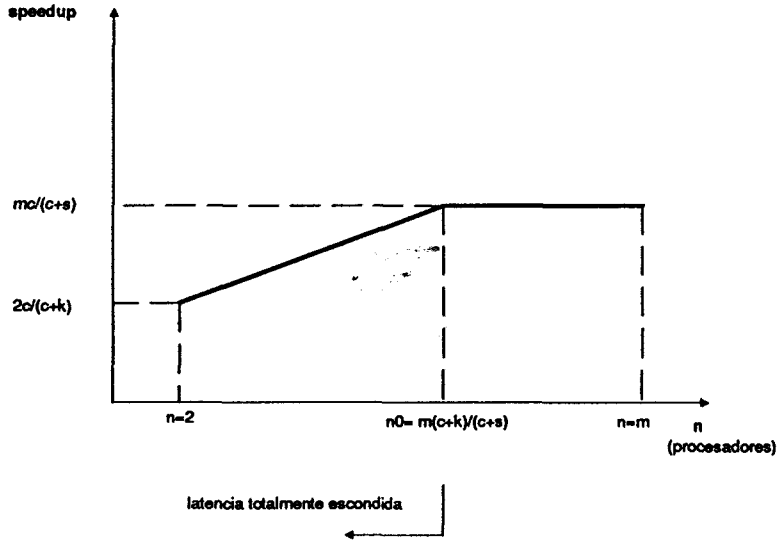

Fig.2.9 *Speedup* **en función de procesadores**

escondida y, por lo tanto, la utilización es constante.

Pasado este umbral el *speedup* permanece constante (la utilización disminuye al mismo ritmo que aumentan los procesadores), por lo tanto no tiene sentido aumentar el número de procesadores.

Heurística del paralelismo

Analicemos la fórmula antes obtenida que relaciona el paralelismo del programa con el paralelismo de la máquina ($m \geq n \cdot (c+s)/(c+k)$).

Si **k** es bastante más pequeño que **c**, lo que debe ser cierto si se desea obtener una buena utilización de la máquina, se puede despreciar quedando la fórmula:

$m \geq n \cdot (1+s/c)$; o bien $(m-n) \cdot c \geq n \cdot s$

Si **m** es bastante mayor que **n** ($m/n >> 1$) la fórmula podría quedar:

$m \cdot c \geq n \cdot s$; con $c >> k$ y $m >> n$

Esta aproximación implica la suposición de que todos los procesos que corresponden a un procesador ($t = m/n$) se usan para esconder la latencia. Sin embargo, sólo **t-1** procesos esconderán la latencia ya que el otro es el que la produjo. Siempre que exista un número suficiente de procesos por procesador ($t >> 1$), esta aproximación será adecuada.

Podemos tomar esta fórmula como una heurística para conocer si un determinado programa se puede ejecutar de forma eficiente en una determinada máquina.

Es interesante resaltar que el lado izquierdo de esta fórmula esta formado por términos relacionados con las características del programa (número de procesos y granularidad), mientras el lado derecho está relacionado con parámetros de la máquina (número de procesadores y latencia). Así, el producto $n \cdot s$ puede interpretarse como la capacidad generadora de huecos de la máquina y el producto $m \cdot c$ como la capacidad de rellenarlos por parte del programa y, por lo tanto, pueden utilizarse para clasificar máquinas paralelas y programas paralelos respectivamente.

Sean dos máquinas paralelas M_1 y M_2 tales que $n_1 \cdot s_1 = n_2 \cdot s_2$ siendo $s_1 > s_2$. El hueco que hay que rellenar en M_1 es mayor, pero le corresponderán más procesos por cada procesador ($m/n_1 > m/n_2$), por lo que la relación entre el hueco a rellenar y el número de procesos por procesador es la misma en ambas máquinas

($\mathbf{m/(n_1 \cdot s_1)=m/(n_2 \cdot s_2)}$). Ambas máquinas ejecutarán eficientemente los mismos programas siempre que el número de procesos del programa sea bastante mayor que el número de procesadores de ambas máquinas.

En el caso de programas paralelos ocurre lo mismo. Sean P_1 y P_2 dos programas tales que $\mathbf{m_1 \cdot c_1}= m_2 \cdot c_2$ siendo $c_1>c_2$. Los procesos del programa P_1 tienen mayor capacidad de rellenar ($c_1>c_2$), pero existen menos procesos por procesador ($\mathbf{m_1/n<m_2/n}$) con lo que la capacidad de relleno es la misma ($\mathbf{m_1 \cdot c_1/n= m_2 \cdot c_2/n}$). Ambos programas ejecutarán eficientemente sobre las mismas máquinas paralelas siempre que el número de procesadores de la máquina sea bastante inferior al número de procesos de ambos programas.

Veamos, a continuación, algunos ejemplos. Una máquina M_1 de 1.000 procesadores y una latencia de 100μs tendrá el mismo producto $\mathbf{n \cdot s}$ que una máquina M_2 con 10 procesadores y una latencia de 10ms (p.ej. procesadores conectados por una red de área local). Así, un programa P_1 con 10.000 procesos y con un tiempo entre comunicaciones de 10μs ejecutará eficientemente en ambas máquinas. Sin embargo, un programa P_2 con 2.000 procesos y un tiempo entre comunicaciones de 50μs, a pesar de tener el mismo producto $\mathbf{m \cdot c}$ que el programa anterior, no ejecutará tan eficientemente sobre la máquina con 1.000 procesadores, ya que no existe suficiente paralelismo en el programa ($\mathbf{m}$ no es suficientemente mayor que $\mathbf{n}$). Hagamos algunos cálculos.

Los productos $\mathbf{m \cdot c}$ y $\mathbf{n \cdot s}$ son iguales para ambas máquinas y para ambos problemas (100.000μs). Por lo tanto, siguiendo la heurística expuesta antes, los dos programas ejecutarán eficientemente en las dos máquinas. Suponiendo que $\mathbf{k}$ vale 1μs en ambas máquinas, vamos a calcular la utilización en cada caso. Primero calcularemos si la latencia queda totalmente escondida ($\mathbf{(m-n) \cdot c+m \cdot k \geq n \cdot s}$) y dependiendo de ello, hallaremos la utilización.

P_1 sobre M_1: Latencia totalmente escondida (100.000=100.000); $\mathbf{U=c/(c+k)}$=0,91;

P_1 sobre M_2: Latencia totalmente escondida (109.900>100.000); **U**= 0,91;

P_2 sobre M_1: Latencia parcialmente escondida (52.000<100.000); **U=t•c/(c+s)**=0,67;

P_2 sobre M_2: Latencia totalmente escondida (101.500>100.000); **U**= 0,91

En el tercer caso (P_2 sobre M_1), no hay suficiente paralelismo en el programa.

La heurística no es válida ya que no se cumple una de las restricciones (**m>>n**) necesarias para su utilización.

Simplificaciones en el análisis

Todo el desarrollo realizado sobre la técnica de multiplexación partía de un modelo simplificado, tanto de los parámetros que caracterizan una máquina paralela como de los de un programa paralelo. A continuación, vamos a revisar estas simplificaciones analizando sus repercusiones.

En primer lugar, la latencia en las comunicaciones depende de la tasa de comunicaciones que exista en el sistema, esto es, la latencia **s** es función del tiempo entre comunicaciones **c**. No tardará lo mismo, evidentemente, un acceso a memoria compartida siendo el único acceso en el sistema que existiendo gran cantidad de accesos en ese instante. Esto es debido a que existirán colisiones en los accesos y deberán ser encolados.

Lo mismo ocurre con un mensaje que atraviesa una red de comunicación. Si la red está bajo condiciones de mucha carga, el mensaje tardará más en llegar a su destino debido a las colisiones y encolamientos.

Esta situación puede llevar a que programas con granularidad muy fina (**c** muy pequeño), aumenten considerablemente la latencia en una determinada máquina al colapsar las comunicaciones. En este caso sería necesario, para calcular la relación óptima entre el número de procesos del programa **m** y el número de procesadores de la máquina **n**, sustituir la latencia constante por una función que dependa de la tasa de comunicaciones del programa.

Existen algunas propuestas, como el *hashing* de memoria y el encaminamiento aleatorio en dos fases, que permiten reducir probabilísticamente las colisiones y encolamientos, en condiciones de alta carga, en los accesos a memoria y en el encaminamiento de mensajes respectivamente. Con estas técnicas se puede conseguir un valor relativamente constante de la latencia con bastante independencia de la tasa de comunicaciones.

Otra simplificación muy importante ha sido suponer que los procesos del programa eran independientes realizando, por lo tanto, solamente interacciones sin sincronización. Bajo esta visión, un proceso ejecuta **c** unidades de tiempo, a continuación inicia una interacción, se produce el cambio de contexto y, aproximadamente **s** unidades de tiempo después, la transferencia está terminada pasando el proceso a estado ejecutable. Para utilizar eficientemente el procesador, durante este tiempo se ejecutan otros procesos que permiten esconder esta latencia.

Las interacciones con sincronización, aunque tratadas de la misma forma, tienen un comportamiento diferente. Así, si un proceso, después de **c** unidades de tiempo ejecutando, inicia una interacción con sincronización, se produce un cambio de contexto como en el caso anterior. Sin embargo, en este caso no se puede conocer a priori el tiempo que tardará la transferencia ni cuantos procesos estarán disponibles para esconder la latencia ya que algunos de ellos pueden estar bloqueados esperando la finalización de otras transferencias con sincronización. Estos valores dependerán de como sean las dependencias de los procesos del programa correspondiente y de como se realice la asignación entre procesos y procesadores.

Un programa está formado por un conjunto de procesos con unas ciertas dependencias entre ellos que deben cumplirse para llevar a cabo el objetivo del mismo. Estas dependencias definen el perfil de paralelismo del programa, esto es, el paralelismo que se podría obtener ejecutándolo en una máquina con infinitos procesadores y una latencia nula. En este perfil se podrían identificar las fases con menor o mayor paralelismo del programa.

Cuando se desea ejecutar un programa en una máquina real (número finito de procesadores y latencia nula), es necesario analizar las dependencias de los procesos del programa para asignarlos de la mejor forma posible. Así, puede ser interesante asignar dos procesos independientes a distintos procesadores o dos procesos que se sincronizan al mismo procesador consiguiendo que la sincronización sea local al procesador.

El análisis de la dependencia entre los procesos para realizar una asignación que proporcione una utilización eficiente de la máquina puede ser muy complejo. Si se utiliza un modelo de dependencias entre procesos bastante restringido, el análisis puede simplificarse considerablemente, aunque con la consiguiente pérdida de expresividad. Así, por ejemplo, Sarkar [SAR89] realiza dicho análisis sobre un modelo de dependencias acíclico en el que, además, una tarea una vez arrancada sólo interactúa con otras al final de su ejecución.

Estas dificultades en la asignación de procesos a procesadores se presentan en el caso de asignación estática (un proceso está asignado al mismo procesador durante toda su existencia), que es lo que hemos supuesto por ser más restrictivo. Si se utiliza asignación dinámica (un proceso a lo largo de su existencia está asociado a distintos procesadores), la correspondencia entre procesos y procesadores se puede realizar en tiempo de ejecución pudiendo seleccionarse cualquier proceso. La asignación dinámica facilita, por lo tanto, la utilización eficiente de la máquina ya que se puede seleccionar para esconder la latencia cualquier proceso que esté ejecutable, y no sólo los asignados estáticamente al procesador. La asignación dinámica, como vimos anteriormente, es factible en máquinas con memoria compartida, pero si no existe memoria compartida implica una migración del proceso entre los procesadores correspondientes. Esta migración no tendrá sentido en la mayoría de los casos, ya que los beneficios de ocupar un procesador con un proceso no compensan la pérdida de eficiencia correspondiente a mover el proceso entre los procesadores.

Como conclusión, podemos decir que la simplificación de considerar todos los procesos independientes, nos permite realizar un análisis de la multiplexación sin

tener en cuenta las características, en cuanto dependencias entre procesos, de un programa en concreto. Digamos que, de todos los posibles programas con el mismo número de procesos y la misma granularidad, elegimos el de máximo paralelismo, que será aquel en el que no hay dependencias entre los procesos ya que éstas disminuyen el paralelismo.

2.4.3 Requisitos para el soporte eficiente de la multiplexación

La utilización provechosa de la técnica de multiplexación en una determinada máquina requiere que la misma proporcione de una forma eficiente una serie de mecanismos. Los principales requisitos que debe proporcionar son:

- Gestión eficiente de los procesos. Se necesita que las operaciones de cambio de contexto y las de creación y destrucción de procesos sean muy rápidas. El tiempo que se tarda en realizar estas operaciones en una determinada máquina, va a acotar la más fina granularidad que puede tener un programa para poder ejecutarse eficientemente en dicha máquina ($k<c$).

- Las transferencias, tanto los accesos a la memoria compartida como el paso de mensajes, deberían ser de ciclo partido. Así, una vez que un proceso inicia una transferencia, se realiza un cambio de contexto y se ejecuta otro proceso. Un procesador, por lo tanto, deberá poder tener múltiples transferencias pendientes simultáneamente y deberá poder manejar las notificaciones de terminación de las transferencias que podrán llegar en cualquier orden. Es importante, además, que el tiempo que el procesador dedica a gestionar los inicios y terminaciones de transferencias sea lo más corto posibles (estos tiempos estaban incluidos en k).

- La red de interconexión, ya sea procesador-procesador o procesador-memoria, debe tener altas prestaciones (número de transacciones por segundo). En caso contrario, un programa con granularidad fina puede, como se expuso antes, colapsar la red.

Esta eficiencia es muy difícil proporcionarla sin apoyo hardware. Así, si la máquina física no proporciona estos mecanismos, es muy difícil implementarlos eficientemente en otros niveles. En la máquina virtual sistema operativo, por ejemplo, la gestión de los procesos implicaría realizar llamadas al sistema, las cuales no proporcionan normalmente la suficiente eficiencia.

Fruto de esas necesidades, han ido apareciendo propuestas de arquitecturas, tanto comerciales como experimentales, que cumplen algunos de estos requisitos.

En el campo comercial, la familia de microprocesadores Transputer de Inmos proporciona algunas de estas características, como por ejemplo los cambios de contexto se realizan en un tiempo por debajo del microsegundo.

En el campo experimental, se pueden destacar algunas propuestas que intentan añadir ideas de las arquitecturas de flujo de datos a las arquitecturas von Neumann.

La arquitectura híbrida de Ianucci [IAN88] y el procesador P-RISC de Nikhil y Arvind [NIK89] son ejemplos de estas propuestas.

Vamos a revisar, a continuación, algunas características del P-RISC analizando como responden a los requisitos expuestos.

P-RISC

El P-RISC (Parallel RISC) es un procesador de tipo RISC diseñado para construir máquinas paralelas formadas por procesadores con memoria local y módulos de memoria accesibles desde todos los procesadores. El código y los operandos de las instrucciones aritméticas que ejecuta un procesador están almacenadas en la memoria local. Siguiendo el estilo RISC, solamente se accede a la memoria compartida mediante instrucciones *load* y *store* que mueven datos entre la memoria compartida y la local. Los datos dentro de la memoria local están organizados en diferentes regiones (*frames*) de forma que cada proceso trabaja sobre un determinado *frame*. Cada *frame* puede verse en cierta forma, como un banco de registros asociado a un determinado proceso.

Las principales características del P-RISC son:

- Proporciona soporte para la gestión de procesos. Su juego de instrucciones incluye instrucciones para crear y terminar procesos (*fork* y *join*). La ejecución de las instrucciones se lleva a cabo siguiendo una filosofía inspirada en la arquitectura de flujo de datos. En cada procesador existe una cola de tokens que identifican las instrucciones listas para ser ejecutadas. Cada token incluye la dirección de la instrucción y la dirección del comienzo del *frame* asociado al proceso. El procesador va tomando tokens de esta cola y va ejecutando las instrucciones correspondientes a través de un pipeline típico (búsqueda de instrucciones, lectura de operandos, ejecución y almacenamiento de resultados). Cada vez que una instrucción normal sale del pipeline, se inserta dentro de la cola un token que apunta a la instrucción almacenada en la siguiente posición de la memoria local. En el caso de un salto, inserta un token que apunta a la instrucción destino de salto.
  Cuando se ejecuta un *fork* se insertan dos tokens, uno correspondiente a la siguiente instrucción y otro que corresponde a la primera instrucción del nuevo proceso. Usando esta técnica se facilita la gestión del pipeline ya que desaparecen las dependencias entre las instrucciones de un mismo proceso.
  Este procesador ejecuta, por lo tanto, simultáneamente (en diferentes fases de pipeline) instrucciones de varios procesos. Esto es posible gracias a que cada instrucción incluye el contexto correspondiente, desapareciendo la operación de cambio de contexto. Como contrapartida, se puede perder eficiencia en las partes secuenciales de un programa ya que una instrucción de un proceso no se comienza a ejecutar hasta que no se termine la instrucción anterior del mismo proceso (como si no existiese pipeline en la máquina).

- Proporciona interacciones de ciclo partido con la memoria compartida, ya sean interacciones con sincronización o sin ella. Existen tres operaciones:

  - *Load* convencional. Cuando se ejecuta esta instrucción se manda un mensaje a la memoria, pero no se inserta ningún token en la cola. Por lo tanto, la

ejecución de este proceso queda bloqueada. Cuando la memoria termine la operación, mandará un mensaje al procesador con el valor correspondiente y éste, al recibirlo, generará un token que apunte a la siguiente instrucción del proceso continuándose la ejecución del mismo.

- *Load* con sincronización. Cada posición de la memoria compartida incluye información de estado formando una estructura denominada *Estructura-I*.
  Si la posición a la que accede *load* está llena, se comporta como un *load* convencional. Si esta vacía, se incluirá el proceso en una lista de lecturas pendientes que existe asociada a cada posición.
  La diferencia entre ambos tipos de *load* se encuentra en el tiempo en que tardará el procesador en recibir el mensaje con el valor accedido. En el primer caso, depende de la latencia de la máquina, mientras que en el segundo depende del estado de ejecución de los procesos del programa.

- *Store.* Cuando se ejecuta esta instrucción se manda a memoria el mensaje correspondiente generándose el token que apunta a la siguiente instrucción del proceso (no hay bloqueo). Cuando se escribe en una posición vacía, la memoria debe mandar mensajes con el nuevo valor a todos los procesos (si existiese alguno) que estuviesen esperando en la lista de lecturas pendientes asociada a dicha posición. Cuando reciban el mensaje los procesadores correspondientes, actuarán de la misma forma que en el caso de un *load* convencional, esto es, generarán el token de la siguiente instrucción.

- En cuanto a la red de interconexión, no se realiza ningún comentario sobre sus características o sus prestaciones ya que la propuesta está centrada en las características del procesador. Sin embargo, hay que recordar que es un punto muy importante en el diseño de una máquina paralela.

2.5 El problema de los bloqueos anidados

En este apartado se va a analizar otro factor que influye en la correspondencia entre niveles de máquinas virtuales: Los bloqueos anidados.

Normalmente, cuando un procesador de nivel N se queda bloqueado esperando un evento, el procesador de nivel N-1 al que está asociado puede pasar a ejecutar otro procesador en estado ejecutable. Pero, en algunas ocasiones, para satisfacer la operación pedida por el procesador de nivel N, el de nivel N-1 debe pedir servicio al de nivel N-2 asociado y quedarse bloqueado a la espera de la finalización del servicio. Así, sucesivamente, pueden quedarse bloqueados procesadores de distintos niveles.

Como consecuencia de este anidamiento de bloqueos el paralelismo del sistema se degradará en mayor o menor grado dependiendo del tipo de correspondencia que exista entre ambos niveles. Suponiendo que se multiplexan **m** procesadores de nivel N sobre **n** procesadores de nivel N-1, se pueden presentar los siguientes casos:

- **m>n (n=1).** El bloqueo del procesador de nivel N-1 tiene como consecuencia que toda la máquina virtual de nivel N queda bloqueada, ya que todos los procesadores de nivel N están asociados al mismo.

- **m>n (n≠1).** Se degrada el paralelismo del sistema, pero no se queda bloqueada toda la máquina de nivel N. Si se trata de una asignación estática se quedarán bloqueados todos los procesadores asociados al procesador de nivel N-1 bloqueado. En el caso de asignación dinámica se degrada el paralelismo de la máquina que en ese momento dispondrá de **n-1** procesadores.

- **m=n.** Sólo se bloquea el procesador de nivel N-1 asociado al procesador que causó el bloqueo, ya que con este esquema hay una correspondencia directa entre los procesadores de ambos niveles.

Se ha elegido el término "problema de bloqueos anidados" que recuerda al ya conocido "problema de llamadas anidadas a monitores" [PET85]. Este problema sur-

ge cuando un proceso dentro de un monitor llama a otro monitor y se queda bloqueado (**wait**) en este último. Por la definición de monitor, el último monitor queda "liberado" y puede ser accedido por otros procesos, pero el primero (y todos excepto el último, si la llamada estaba anidada a más de un nivel) queda bloqueado. Como se puede ver las características de ambos problemas son bien diferentes, aunque su efecto tiene alguna similitud, disminuyen la concurrencia del sistema.

En esta sección se han considerado tres factores que influyen fuertemente en la forma de implementar una determinada máquina virtual sobre otra ya existente, a saber, el modelo de interacción al que responden ambas máquinas, la granularidad del paralelismo de las mismas y el problema de los bloqueos "anidados". Pero existen, en cada caso concreto, otros factores que hay que tener muy en cuenta a la hora de implementar la máquina correspondiente. En el siguiente capítulo se estudiarán las características específicas de la "máquina Ada".

2.6 Sumario del capítulo

Una buena herramienta para programar computadores paralelos debe ser independiente de la máquina pero, al mismo tiempo, debe poder implementarse eficientemente en muy diferentes tipos de máquinas. Con el cumplimiento de este requisito se facilita enormemente la portabilidad del software paralelo, así como la labor del programador.

El objetivo de este capítulo ha sido analizar si es posible el cumplimiento de dicho requisito, esto es, si se pueden realizar programas independientes de la máquina que puedan ejecutar eficientemente en muy diferentes clases de máquinas.

Este análisis ha partido de la definición de un modelo abstracto de un sistema paralelo. En este modelo un sistema está estructurado como una jerarquía de máquinas virtuales paralelas, donde cada máquina virtual queda definida sobre la de nivel inferior por un Entorno de Ejecución que se encarga de proporcionar la funcionalidad de dicha máquina a partir de la proporcionada por la de nivel inferior.

A pesar de la gran variedad de características que puede tener una máquina virtual paralela, hemos resaltado dos de ellas: El modelo de interacción entre los procesadores de la máquina virtual y la asignación entre los procesadores de la máquina y los procesadores de nivel inferior. Se han distinguido tres modelos de interacción, a saber, memoria compartida, memoria distribuida y un modelo mixto.

En cuanto a la asignación entre procesadores, nos hemos interesado en la relación entre el número de procesadores de la máquina superior con respecto a los de la inferior, y en como se realiza dicha asignación, estática o dinámicamente.

Este modelo refleja el concepto de independencia, el programador sólo necesita conocer las características de la máquina virtual con la que trabaja. Pero, ¿puede implementarse eficientemente la máquina virtual seleccionada sobre la máquina física? En algunos casos esto puede ser difícil, por ejemplo implementar una máquina virtual con un modelo de memoria compartida sobre una máquina con memoria distribuida. Más allá, suponiendo una máquina que pueda ser implementada con eficiencia, puede ocurrir que un determinado programa no ejecute eficientemente.

Esta problemática ha sido analizada bajo el concepto de granularidad. La granularidad proporcionada por una máquina expresa su capacidad de comunicación frente a la de procesamiento. La granularidad de un programa, por su parte, refleja las necesidades de comunicación del mismo. Si estos valores no son adecuados, el programa no ejecutará eficientemente en la máquina.

Un programador debe, por lo tanto, seleccionar una máquina virtual que pueda ser implementada eficientemente en el computador que va utilizar y, además, codificar el programa de forma que su granularidad sea compatible con la de la máquina.

Frente a este escenario, se ha expuesto la técnica de utilizar el exceso de paralelismo (mayor número de procesos que de procesadores) para esconder la latencia.

La utilización poco eficiente de las máquinas paralelas se debe a los intervalos de tiempo en los que el procesador no realiza trabajo útil debido a que está esperando que se termine una transferencia (p.ej. un acceso a memoria compartida) o a que se

produzca una sincronización (p.ej. la recepción del mensaje). Con el exceso de paralelismo se pretende rellenar este tiempo ejecutando otros procesos mientras tanto. Teniendo el suficiente número de procesos y un cambio de contexto muy rápido, se puede conseguir una buena utilización de la máquina. Se ha realizado un análisis cuantitativo de este método obteniendo varios resultados, entre ellos heurísticas que permiten clasificar tanto las máquinas paralelas como los programas paralelos. Se ha planteado también la necesidad de que esta técnica tenga soporte hardware para poder llevarla a cabo de una forma eficiente.

Utilizando el exceso de paralelismo se puede implementar de forma eficiente una máquina virtual con memoria compartida sobre una máquina con memoria distribuida, siempre que exista el suficiente exceso de paralelismo y que, cuando se produzca un acceso a la memoria compartida, se bloquee el proceso y se realice un cambio de contexto. De esta forma se llega a una relativa unificación entre los dos paradigmas clásicos: Memoria compartida y memoria distribuida. Esta unificación, sin embargo, no es aplicable a la asignación entre procesadores, sólo se puede realizar directamente una asignación dinámica si la máquina física proporciona memoria compartida.

La labor del programador consistirá en realizar un algoritmo que contenga el máximo paralelismo posible. Si el programa no ejecuta eficientemente en la máquina destino, no debe modificar el programa sino replantear la utilización de la máquina usando, por ejemplo, menos procesadores para ejecutar el programa.

Se ha analizado, por último, el problema de los bloqueos anidados y la repercusión del mismo dependiendo del tipo de asignación entre procesadores utilizada.

3

Niveles de paralelismo en un sistema. Modelo de paralelismo de Ada

3.1 Introducción

En la primera parte de este capítulo se realiza una aplicación a sistemas reales del modelo abstracto jerárquico presentado en el capítulo anterior. Esta aplicación permite distinguir tres niveles de máquinas virtuales en un sistema: La máquina física, la máquina definida por el sistema operativo y la máquina definida por un lenguaje paralelo.

En un sistema pueden existir otros niveles de paralelismo. Por ejemplo, el procesador P-RISC descrito en el capítulo anterior, presenta una máquina virtual "arquitectura" por encima de la máquina física. Sin embargo, nos centraremos en los tres niveles típicos.

En cada uno de los niveles se distinguirán diferentes tipos de máquinas presentándose ejemplos de las mismas. Se realizarán también algunas consideraciones sobre el paralelismo síncrono que, aunque fuera de nuestro modelo, puede incluirse de alguna forma en el mismo.

En la última parte se exponen algunas de las características principales del modelo de paralelismo de Ada comparándolo con otros lenguajes paralelos. Por último, se revisan la terminología y las características generales del Entorno de Ejecución de Ada (ARTE: *Ada Runtime Environment*).

3.2 Niveles de paralelismo y concurrencia en un sistema convencional

Existen, típicamente, tres niveles de concurrencia en un sistema convencional:

- **Nivel Físico.** Proporciona paralelismo mediante la utilización de múltiples procesadores. Existen tres modelos de interacción entre los procesadores:

  - Máquinas con memoria compartida. Dentro de este grupo se incluyen arquitecturas con memoria compartida convencionales UMA (UMA: Tiempo de acceso a memoria uniforme), arquitecturas NUMA (NUMA: Tiempo de acceso a memoria no uniforme) y sistemas monoprocesadores como un caso particular.

    - Arquitecturas UMA (también llamadas multiprocesadores simétricos). Existe una memoria común a todos los procesadores. Los problemas inherentes a este tipo de arquitecturas son los choques en los accesos a la memoria compartida y su dificultad de crecimiento. En estas máquinas, por lo tanto, existen un número relativamente reducido de procesadores. Suelen existir memorias cache para intentar minimizar los accesos a la memoria compartida. El algoritmo utilizado para mantener la coherencia entre las caches y la memoria principal es muy importante para el rendimiento del sistema. En este tipo de máquinas se puede realizar directamente una asignación dinámica ya que el código de cualquier proceso puede ser accedido por cualquier procesador. Ejemplos de este tipo de máquinas son: Multimax de Encore (máximo 20 procesadores), Balance de Sequent (máx. 30) y las series FX de Alliant (máx. 8) [GEHR88].

Existen algunas propuestas para construir arquitecturas de este tipo que no presenten los problemas de crecimiento antes descritos. El ParaDiGM [CHE91] propone utilizar una jerarquía de caches, entre otras técnicas, para evitar estos problemas.

- Arquitecturas NUMA. En estas máquinas no existe una memoria global sino que cada procesador tiene su memoria asociada. Pero dicha memoria, a diferencia de las máquinas sin memoria compartida, acepta peticiones de acceso desde otros procesadores. Con este esquema se consiguen sistemas con mejor capacidad de crecimiento que en el caso anterior y, además, desaparece la memoria global como cuello de botella del sistema. Pero hay que tener en cuenta las repercusiones que tiene la diferencia en el tiempo de acceso entre las peticiones de un procesador a su memoria local y las peticiones a memorias remotas. Debido a esta diferencia no es posible realizar directamente una asignación dinámica. En este tipo de arquitecturas, para realizar una asignación de ese tipo, es necesario realizar una migración del proceso correspondiente hacia la memoria local del procesador.

  Esta situación lleva a algunos autores a no incluir este tipo de sistemas como máquinas con memoria compartida. Un ejemplo de este tipo de máquinas es el sistema BBN Butterfly (hasta 256 procesadores). En este sistema una petición a una memoria local tarda alrededor de 500 ns. mientras que una petición remota 5 μs.

- Máquinas sin memoria compartida. Una máquina de este tipo está estructurada como un conjunto de nodos (un procesador y su memoria local como mínimo) conectados por una red de interconexión. Dentro de este tipo se van a incluir sistemas de muy diferentes características: Desde multiprocesadores débilmente acoplados, hasta computadores conectados por una red

de área local (LAN) o incluso redes de mayor alcance (WAN). Algunos ejemplos son:

  - Multiprocesadores cuya red de interconexión está basada en un hipercubo, como el iPSC de Intel o el N-Cube.

  - Máquinas formadas por un conjunto de microprocesadores con memoria local conectados por una red de área local. Por ejemplo un conjunto de procesadores de la familia M68000 conectados por una red Ethernet. La comunicación en este tipo de máquinas es considerablemente más lenta que, por ejemplo, en máquinas basadas en hipercubos.

- Máquinas con un modelo mixto. Una máquina de este tipo esta formada, como en el caso anterior, por un conjunto de nodos conectados por una red de interconexión. Sin embargo, dentro de cada nodo pueden existir múltiples procesadores que comparten memoria.

Dentro del nivel físico no se ha considerado el paralelismo existente en máquinas vectoriales y máquinas SIMD, ya que el modelo propuesto tiene en cuenta únicamente el paralelismo introducido por la ejecución simultánea de múltiples flujos de instrucciones independientes. El modelo, por lo tanto, está orientado hacia el paralelismo *asíncrono*, no contemplando el paralelismo *síncrono* [DUN90].

No se han considerado tampoco arquitecturas que siguen modelos de ejecución no convencionales, como las arquitecturas de flujo de datos (*data-driven*) o las arquitecturas de reducción (*demand-driven*), ni arquitecturas de propósito especial, como arquitecturas sistólicas o las de frente de onda (*wavefront array architectures*).

En la clasificación no se ha diferenciado entre sistemas *homogéneos* y sistemas *heterogéneos*. En un sistema *homogéneo* todos los procesadores son del mismo tipo. En un sistema *heterogéneo* los procesadores son de diferentes tipos. Los

sistemas *heterogéneos* presentan problemas debido a que cada procesador puede tener diferente manera de representar las instrucciones y los datos. Sin embargo, estos problemas, aunque importantes, no conciernen directamente al modelo propuesto.

- **Nivel de Sistema Operativo.** El S.O. proporciona concurrencia multiplexando la ejecución de los procesos existentes en los diferentes procesadores físicos. En este nivel se pueden distinguir también los tres modelos de interacción.

  - Sistemas Operativos sin memoria compartida. En general, los Sistemas Operativos multiusuario, por razones de protección y seguridad entre los usuarios, implementan procesos con espacio de direcciones disjuntas. Los procesos se comunican entre sí mediante algún mecanismo de paso de mensajes.

  - Sistemas Operativos que siguen un modelo de memoria compartida:

    - Algunos S.O., a pesar de proporcionar mapas de direcciones disjuntas para los procesos, incluyen facilidades para compartir zonas de memoria entre procesos. Por ejemplo, UNIX 4.3BSD incluye la llamada al sistema **mmap()**, y VMS incluye el mecanismo **Global Sections**. Estos mecanismos, además de para compartir datos entre procesos, suelen utilizarse para compartir código (bibliotecas compartidas). En [PER87] se describe un ejemplo de la utilización de **Global Sections** para compartir datos entre procesos.

    - En algunos S.O., sobre todo monousuario, se proporciona un espacio de direcciones común para todos los procesos, dejando que sea el tipado del lenguaje de programación correspondiente el que proporciona la protección necesaria entre los distintos procesos. El entorno de programación Cedar [COU88] y el Sistema Pilot [KEE85], basa-

dos en el lenguaje concurrente Mesa, son ejemplos de este tipo de sistemas.

- Sistemas Operativos que siguen un modelo mixto. Una tendencia en el campo de los S.O., sobre todo en S.O. distribuidos, es reemplazar el concepto convencional de proceso por dos tipos de procesos: Procesos "pesados" (*heavy-weigth process*) y procesos "ligeros" o flujos de ejecución (*ligth-weigth process* o *thread*)[FEI90][COU88]. Los procesos pesados tienen contextos independientes (mapas de memoria disjuntos) proporcionando protección y seguridad. Dentro del contexto de un proceso pesado pueden existir múltiples procesos ligeros que, generalmente, comparten el espacio de direcciones. Por lo tanto, la comunicación entre procesos ligeros (que son los procesadores virtuales de esta máquina S.O.) pertenecientes al mismo proceso pesado puede realizarse mediante memoria compartida, mientras que si los procesos ligeros están asociados a diferentes procesos pesados (contextos) se comunican usando algún mecanismo de paso de mensajes. Ejemplos de este tipo de sistemas son: Mach, V-Kernel, Amoeba, Chorus, SOS.

- **Nivel de usuario.** Se puede introducir concurrencia en un sistema mediante un lenguaje concurrente. En este nivel también pueden presentarse tres modelos de interacción:

  - Máquina sin memoria compartida. Algunos lenguajes concurrentes, como CSP, Occam o NIL no permiten que los procesos (término genérico para nombrar a las entidades concurrentes del lenguaje correspondiente) compartan memoria. La comunicación entre procesos se realiza mediante el mecanismo de paso de mensajes.

  - Máquina con memoria compartida. Existen lenguajes concurrentes, como Ada y Mesa, que permiten que los procesos compartan memoria.

En Concurrent C [GEH89], por otro lado, no es obligatorio que una implementación proporcione memoria compartida entre los procesos. Así, aunque Concurrent C no prohibe que un proceso referencie una variable global, no se asegura que el funcionamiento del programa sea correcto en cualquier implementación. La máquina virtual definida por Concurrent C sigue un modelo sin memoria compartida, aunque implementaciones particulares pueden proporcionar dicha memoria compartida.

- Máquina con un modelo mixto. El lenguaje SR (*Synchronizing Resources*) [BAL89], por ejemplo, presenta este modelo de interacción. Un programa en SR está formado por un conjunto de módulos llamados "recursos" (*resources*), dentro de los cuales pueden definirse múltiples procesos. Los procesos pertenecientes a un mismo "recurso" comparten memoria, mientras que los definidos en distintos "recursos" no comparten memoria y utilizan para su comunicación mecanismos de paso de mensajes.
  La introducción del concepto Nodo Virtual (*Virtual Node*) en lenguajes concurrentes como Ada define también, como se analizará más adelante, un modelo mixto de interacción.

Dentro de este nivel se considera únicamente la concurrencia que el programador introduce explícitamente: Es el programador el encargado de definir las actividades que pueden ejecutarse potencialmente en paralelo. No se considera concurrencia de usuario el paralelismo implícito que puede existir en un programa y que alguna herramienta puede extraer del mismo. Por ejemplo, el código generado por un compilador para la ejecución paralela de un bucle no introduce concurrencia de usuario.

Black [BLA90] plantea una clasificación de los modelos de concurrencia dependiendo de las relaciones entre los tres niveles estudiados. La tabla 3.1 es una adaptación de la clasificación de Black. Existen cuatro categorías:

PU=PS=PF	Paralelismo puro
PU=PS≥PF	Concurrencia de S.O.
PU≥PS=PF	Concurrencia de usuario
PU≥PS≥PF	Concurrencia dual

PU	Procesador virtual del nivel de usuario
PS	Procesador virtual del nivel S.O.
PF	Procesador físico

Tabla 3.1 Niveles de paralelismo y concurrencia

- *Paralelismo Puro.* No se introduce concurrencia sobre el paralelismo físico. El grado de concurrencia de la aplicación está limitado por el número de procesadores físicos existentes.

- *Concurrencia de S.O.* El S.O. introduce concurrencia sobre el paralelismo físico. Es éste el tipo de concurrencia visible por el usuario del S.O.

- *Concurrencia de Usuario.* Un lenguaje concurrente introduce concurrencia sobre el paralelismo físico. La utilización de un lenguaje de este tipo sobre una máquina "desnuda" (sin S.O.) sería un ejemplo de este tipo de concurrencia

- *Concurrencia Dual.* Se introduce concurrencia en ambos niveles, S.O. y usuario. El S.O introduce concurrencia sobre el paralelismo físico y, a su vez, un lenguaje concurrente lo hace sobre la "máquina S.O.".

Por último, se exponen algunas consideraciones sobre el paralelismo y concurrencia síncronos. Este tipo de paralelismo, como se vio anteriormente, no está englobado dentro del modelo propuesto, pero puede ser interesante realizar algunos comentarios sobre el mismo.

Dentro del paralelismo síncrono también se pueden distinguir diferentes niveles de concurrencia y paralelismo. A continuación se expone un ejemplo en el que se distinguen dos niveles de concurrencia:

- Nivel físico. Una máquina con arquitectura SIMD como la Máquina de Conexión de Thinking Machines. Este sistema está formado por un gran número de procesadores (hasta 65.536) de un bit organizados en conjuntos de 16 procesadores. Cada conjunto se conecta con el resto en una topología hipercubo.

- Nivel de usuario. El lenguaje C* presenta un modelo de concurrencia síncrono (*data-parallel*). Este lenguaje fue diseñado para programar la Máquina de Conexión. C* es una extensión de C orientada al procesamiento de múltiples datos en paralelo. C* permite definir datos de tipo paralelo (**poly**). Un programa C* es un programa secuencial, pero cuando un operador se aplica a un tipo de datos paralelo, se realizan en paralelo las operaciones sobre todas las instancias de ese dato.

  El lenguaje C* define una máquina virtual síncrona en la que los procesadores virtuales son los elementos de procesamiento necesarios para operar en paralelo sobre todas las instancias de un dato, independientemente del número de elementos de procesamiento que proporcione la máquina física.

Se podría generalizar, por lo tanto, el modelo propuesto para que incluyera este tipo de máquinas virtuales. De esta forma, en cada nivel podrían existir cuatro clases de máquinas: Máquinas síncronas, máquinas asíncronas sin memoria compartida, máquinas asíncronas con memoria compartida y máquinas asíncronas mixtas.

Habría que generalizar también el estudio de la problemática de implementar una máquina virtual sobre otra que pertenece a una clase diferente para que incluyese a las máquinas síncronas. En relación con esta cuestión, en [QUI90] se recogen las experiencias de implementar C* sobre una máquina "multicomputador" (en la clasificación propuesta una máquina asíncrona sin memoria compartida). La solución se basa en convertir el programa C* del usuario en un programa estilo SCMD (*Same-Code Multiple-Data*). Así, todos los procesadores de la máquina asíncrona ejecutarán el mismo código pero sobre datos diferentes sincronizándose adecuadamente para emular el comportamiento del programa C*.

Se puede resaltar, como conclusión, que idealmente la jerarquía de máquinas virtuales proporciona al usuario independencia de la máquina subyacente. De esta forma, el usuario puede elegir, al diseñar una aplicación, el lenguaje concurrente cuyo modelo sea más adecuado, sin importarle las características de los niveles inferiores de la jerarquía. Así, el usuario puede elegir un lenguaje síncrono para diseñar una aplicación de procesamiento de señales y un lenguaje asíncrono sin memoria compartida para una aplicación de gestión de base de datos, siendo idénticos, en ambos casos, los niveles inferiores de la jerarquía. Esta situación es hipotética ya que, como se vio en el capítulo anterior, no es posible realizar un programa eficiente sin tener en cuenta las características de la máquina física a no ser que se utilice la técnica de usar el exceso de paralelismo para esconder la latencia. Con esta técnica se puede conseguir una relativa independencia.

3.3 Modelo de concurrencia definido por el lenguaje Ada

Los lenguajes concurrentes permiten al programador expresar, mediante múltiples flujos de ejecución, la concurrencia existente entre las distintas actividades de un programa. Un programa concurrente, siguiendo el modelo descrito por Dijkstra [DIJ68], está formado por un conjunto de procesos secuenciales (único flujo de ejecución) autónomos que cooperan entre sí. Aunque la práctica totalidad de los lenguajes concurrentes incorporan el concepto de proceso, existen importantes variaciones en los modelos de concurrencia que adopta cada lenguaje.

En [BUR90], se plantean una serie de características que diferencian los modelos de concurrencia de los diversos lenguajes. A continuación se revisan estas características, enmarcando el modelo de Ada dentro de ellas:

- Estructura:
  - *Estática*: Número de procesos fijo, por lo tanto, conocido en tiempo de compilación. Un ejemplo es el lenguaje Occam.

- *Dinámica*: Los procesos se pueden crear en cualquier momento. El lenguaje Ada sigue este modelo permitiendo que se creen un número dinámico de tareas (los procesos de Ada) ya sea utilizando un tipo acceso a tarea, o bien una formación de tareas cuyo límite sea dinámico.

- Nivel de anidamiento

  - *Anidado*. Se permite definir un proceso dentro de otro proceso. Aparece, por lo tanto, una jerarquía de dependencia entre procesos. Esta jerarquía influye en la activación y finalización de procesos. Ada sigue este modelo. En Ada una tarea depende de la unidad (procedimiento, bloque, tarea o paquete de biblioteca) que crea esa tarea. Las tareas creadas a partir de un tipo acceso a tarea, sin embargo, dependen de la unidad donde está definido el tipo acceso. La unidad de la que depende una tarea se denomina "master" en terminología Ada.

  - *Plano*. No se permite que exista anidamiento. Por ejemplo, Concurrent Pascal.

- Granularidad del paralelismo. Como vimos en el capítulo anterior este concepto refleja las necesidades de comunicación respecto a las de procesamiento de un determinado programa. Cuando se aplica a un lenguaje representa la granularidad típica de los programas escritos en dicho lenguaje.
  El lenguaje Occam 2 tiene una granularidad más fina que Concurrent C ya que los procesos en Occam 2 se comunican normalmente con mayor frecuencia.

- Paso de valores iniciales a un proceso. Existen dos alternativas:

  - Pasar la información como parámetros. Este método es utilizado por el Concurrent C.

  - Comunicarse explícitamente con el proceso creado para pasarle la información inicial. Ada presenta este modelo.

- Terminación de un proceso. Los diferentes lenguajes concurrentes presentan una gran variedad de alternativas en este tema. Algunas de las características que hay que tener en cuenta son las siguientes: La relación entre la jerarquía de procesos y la terminación de los mismos, la existencia de mecanismos que permitan a un proceso "abortar" la ejecución de otro proceso y la posibilidad de que un proceso termine cuando "ya no sea necesario".

  El lenguaje Ada presenta un modelo de terminación de procesos muy completo. La terminación de una tarea se realiza en dos fases. Primero, pasa a un estado de completada y después pasa a terminada. Una tarea pasa al estado completada cuando termina de ejecutar las instrucciones incluidas en su cuerpo. Una tarea completada pasa al estado de terminada cuando todas las tareas dependientes de ella están terminadas. Si no tiene tareas dependientes pasa directamente de completada a terminada. Las tareas dependientes de un paquete de biblioteca son una excepción a esta regla ya que el lenguaje no especifica cuando deben terminar estas tareas.

  La alternativa **terminate** de la sentencia **select** permite que una tarea termine cuando ya no sea necesaria. Una tarea que está esperando en un **select** con una alternativa **terminate** termina cuando se cumplen las siguientes condiciones: La tarea depende de una unidad que está completada y todas las tareas que dependen de esa unidad han terminado o están esperando en un **select** con una alternativa **terminate**. Además de esa tarea terminan todas las tareas que dependen de dicha unidad.

  Una tarea puede abortar la ejecución de otra mediante la sentencia **abort.**

- Representación de un proceso. Existen tres modelos básicos:

  - Modelo basado en **fork** y **join**. No existe una representación explícita de proceso. Cuando se invoca la primitiva **fork**, una rutina comienza a ejecutarse concurrentemente con la rutina que invocó la primitiva. La primitiva **join** permite a la rutina que invocó el **fork** esperar por la finalización de la rutina que ejecuta concurrentemente. El lenguaje Mesa sigue este modelo.

- Modelo basado en **cobegin**. Esta primitiva permite expresar que un conjunto de sentencias se ejecutan concurrentemente. Occam 2 adopta este modelo de concurrencia.

- Declaración explícita de procesos. Este modelo permite expresar la concurrencia dentro de un programa de una forma más clara. Dentro de los lenguajes que siguen este modelo, existen algunos, como Concurrent C, en los que la creación de los procesos debe ser explícita. En otros, como Ada, la activación de los procesos es implícita: Una tarea estática se activa inmediatamente después de la elaboración de la parte declarativa de la unidad en la que se declara la tarea. Una tarea creada mediante un asignador se activa al evaluar el asignador.

- Interacción entre procesos. En este concepto se engloba la sincronización y comunicación entre procesos. Existen dos modelos:

  - Interacción basada en memoria compartida. Los procesos comparten memoria y utilizan algún mecanismo de sincronización para controlar el acceso a la misma. Algunos ejemplos de mecanismo de sincronización son: Semáforos, regiones críticas condicionales (utilizadas en el lenguaje Edison) y monitores (usados en el lenguaje Mesa). El lenguaje Ada permite también este tipo de interacción.
    Las tareas Ada comparten memoria entre sí, controlando el acceso simultáneo a los datos compartidos mediante el mecanismo de comunicación de Ada (el *rendezvous* o cita).

  - Interacción basada en paso de mensajes. Hay una gran variedad de modelos en este tipo de interacción. Estos modelos se diferencian en estas características:

    - La forma en que se sincronizan los procesos que interaccionan.

- Modelo asíncrono o sin espera. El proceso que manda el mensaje continúa ejecutando inmediatamente sin esperar que el proceso receptor acepte el mensaje. El lenguaje Concurrent C soporta este tipo de paso de mensajes.

- Modelo síncrono (*rendezvous*). El proceso que manda el mensaje continúa su ejecución cuando el mensaje ha sido recibido. Occam 2 sigue este modelo.

- Invocación remota (*extended rendezvous*). El proceso que manda el mensaje no continúa su ejecución hasta que le llega una respuesta del proceso receptor. Este mecanismo permite el paso bidireccional de información entre los procesos. El *rendezvous* de Ada sigue este modelo de interacción.

• La forma en que un proceso se refiere al otro. Se pueden hacer dos consideraciones: Dirección y simetría.

  - Esquema directo. El proceso remitente nombra al proceso receptor.

  - Esquema indirecto. El proceso remitente nombra una entidad intermedia (buzón, canal, etc.).

  - Esquema simétrico. El remitente y el receptor se nombran mutuamente (directa o indirectamente).

  - Esquema asimétrico. El receptor no nombra al remitente ni a ninguna entidad intermedia. Este esquema es apropiado para el modelo cliente-servidor.

  Ada, por ejemplo, utiliza un esquema directo asimétrico. Occam, en cambio, un esquema directo simétrico.

Una vez enmarcado el modelo de concurrencia de Ada, en el siguiente capítulo se estudiará la influencia de las características de dicho modelo en la implementación de la "máquina Ada". Antes, en la próxima sección, se revisarán los conceptos y la terminología básica relacionados con el Entorno de Ejecución de Ada (*Ada Runtime System*).

3.3.1 El Entorno de Ejecución de Ada

El Entorno de Ejecución de un lenguaje concurrente, como se vio anteriormente, debe proporcionar la funcionalidad requerida por dicho lenguaje. Normalmente el Entorno de Ejecución forma parte del código generado por el compilador y montador, pero existen otras alternativas. El Entorno de Ejecución de Occam 2 ejecutado sobre un Transputer, por ejemplo, no forma parte del código del programa Occam, sino que está incluido, en microcódigo, dentro del Transputer. Se trata, por lo tanto, de un caso similar al del procesador P-RISC, en el que por encima de la máquina física existe una máquina virtual paralela en el nivel de la arquitectura.

A pesar de la gran variedad de alternativas que se pueden seguir al implementar el Entorno de Ejecución de un determinado lenguaje, puede ser interesante crear un modelo de referencia y una terminología común sobre el Entorno de Ejecución del lenguaje para facilitar la evaluación y comprensión de cada implementación. En el caso del Ada existe un grupo de trabajo, dentro del SIGAda de ACM, encargado de estudiar los temas concernientes al Entorno de Ejecución (RTE: *Runtime Environment*). Este grupo es el ARTEWG (*Ada Runtime Environment Working Group*). El ARTEWG posee una serie de subgrupos que estudian temas específicos del Entorno de Ejecución. El subgrupo 1, por ejemplo, estudia las implicaciones de un Sistema de Tiempo Real Crítico en el Entorno de Ejecución.

Una de las labores del ARTEWG es definir la terminología común que facilita la comunicación entre los diversos grupos interesados en el tema (implementadores, usuarios, etc). En [ART88] se define el Entorno de Ejecución como la unión de tres elementos:

- *Estructuras abstractas de datos*: Convenciones elegidas por la implementación para la representación de los datos.

- *Convenciones en las secuencias de código*: En las llamadas a rutinas y paso de parámetros, en la manipulación de los datos, en el uso de registros, etc.

- *Conjunto de rutinas predefinidas*: A partir del programa fuente (conjunto de unidades) el sistema de compilación crea el "programa Ada generado"; éste incluye los dos primeros elementos del Entorno de Ejecución: Estructuras de datos y secuencias de código. Pero, además, puede hacer falta incluir algunas rutinas predefinidas que soporten características que el implementador no proporciona en el código generado. Estas rutinas que se incluyen son el Sistema en tiempo de Ejecución (RTS: *Runtime System*), y el conjunto de rutinas predefinidas de las que se extrae el Sistema en tiempo de Ejecución se denomina Biblioteca en tiempo de Ejecución (RTL: *Runtime Library*). En la figura 3.1 se representan los distintos elementos del Entorno de Ejecución.

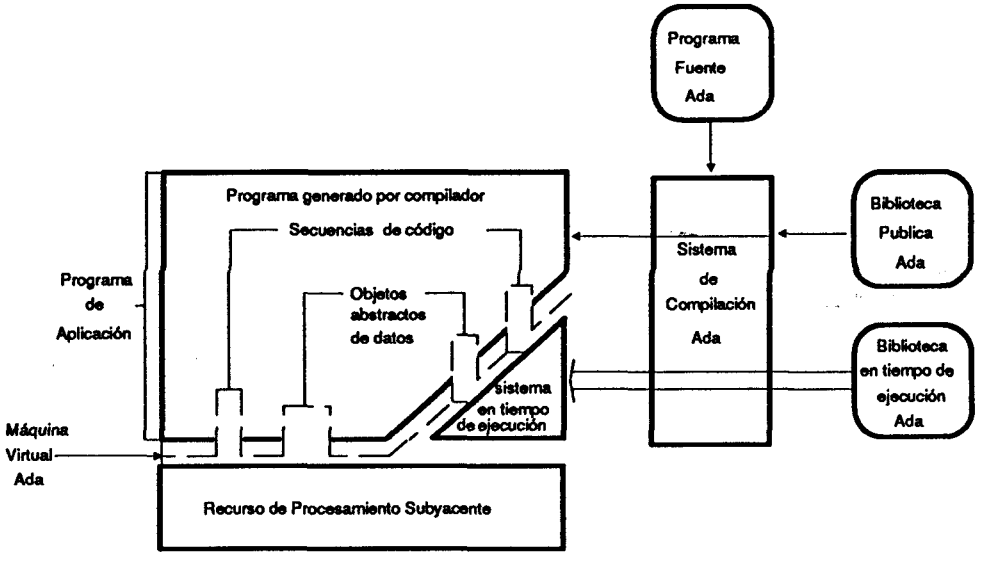

Fig. 3.1 Estructura de un sistema de compilación Ada

Ada está diseñado para su utilización en dos posibles entornos:

- Máquina con sistema operativo o ejecutivo (el S.O. va asociado a máquinas de propósito general y el Ejecutivo a sistemas empotrados). Ver figura 3.2.

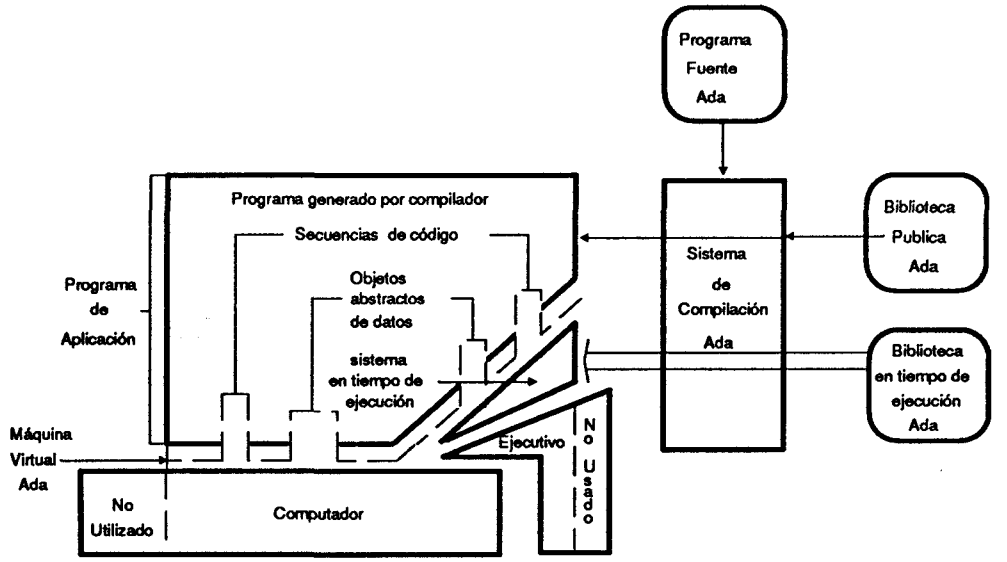

Fig. 3.2 Sistema de compilación para máquinas con S.O.

- Máquinas "desnudas". Ver figura 3.3.

En el primer caso, el Entorno de Ejecución se puede apoyar en algunos de los servicios proporcionados por el S.O. o Ejecutivo para llevar a cabo su función (concurrencia dual: tres niveles de máquinas virtuales paralelas). En el segundo caso, el Entorno de Ejecución debe proporcionar toda la funcionalidad requerida (concurrencia de usuario: dos niveles de máquinas virtuales paralelas).

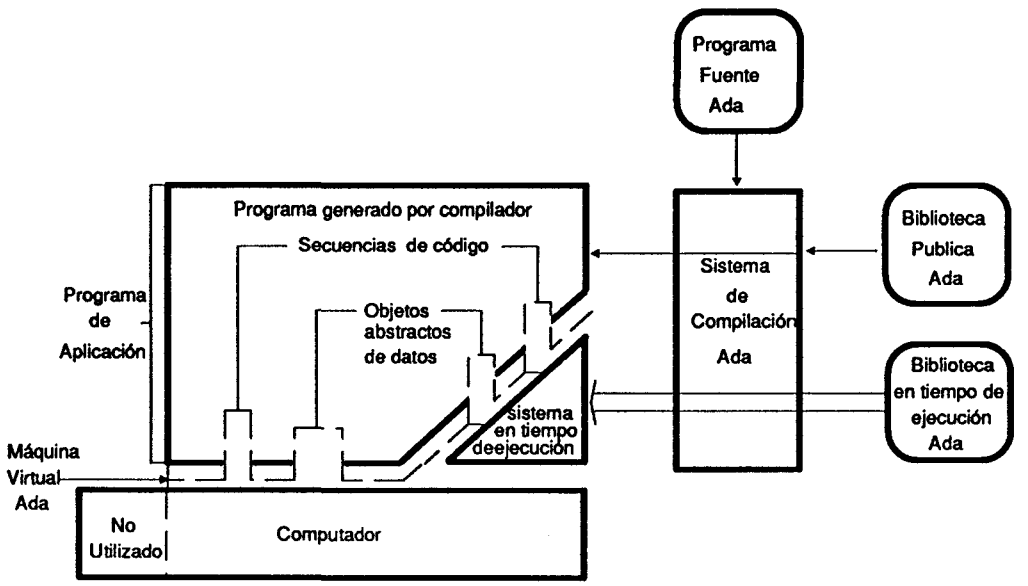

Fig.3.3 Sistema de compilación para máquinas desnudas

Otra labor importante del ARTEWG es la definición de un modelo de referencia para el Entorno de Ejecución. Las convenciones en estructuras de datos y secuencias de códigos no han intentado ser modeladas ya que son muy específicas de cada implementación. El ARTEWG se ha limitado a definir un modelo de una parte del Sistema en tiempo de Ejecución, de las funciones "ejecutivas". Estas funciones corresponden a los servicios que tradicionalmente proporciona un S.O. o Ejecutivo: Gestión de tareas, Gestión de tiempo, Gestión de memoria, Gestión de interrupciones, etc.

3.4 Conclusiones

Como conclusión de este capítulo se podrían resaltar dos ideas:

- Se han encontrado ejemplos de máquinas que siguen los tres modos de interacción (memoria compartida, memoria distribuida y modelo mixto) en los tres niveles analizados. Esto refleja las analogías que existen entre las máquinas virtuales de diferentes niveles.

- El paralelismo síncrono, a pesar de no ser tratado directamente por nuestro modelo, también responde a una estructura de niveles de máquinas virtuales. Podría definirse, por lo tanto, un modelo integrado en el que existieran cuatro posibles modos de interacción: Modelo síncrono, modelo asíncrono con memoria compartida, modelo asíncrono con memoria distribuida, modelo asíncrono mixto.

4

Consideraciones sobre el modelo de paralelismo de Ada

4.1 Introducción

Después del estudio general sobre la implementación eficiente de máquinas virtuales realizado en los anteriores capítulos, en éste nos vamos a centrar en la máquina virtual que define el lenguaje Ada.

Se analizará la granularidad que necesita la máquina Ada, no sólo debida a las interacciones explícitas entre las tareas, sino también a las interacciones implícitas que aparecen como consecuencia de la realización de algunos mecanismos del lenguaje.

Se revisarán las siguientes características:

- El modelo de interacción entre tareas de Ada.
- El anidamiento de tareas y las reglas de ámbito.
- La terminación de las tareas.
- La implementación de las llamadas condicionales y temporizadas.

4.2 Modelo de interacción entre tareas de Ada

El lenguaje Ada proporciona dos mecanismos que permiten la interacción entre tareas: La cita y las variables compartidas.

- La cita o *rendezvous*. Se trata de un mecanismo de interacción con sincronización. Normalmente, por lo tanto, cuando una tarea inicia una cita se queda bloqueada y el procesador correspondiente pasa a ejecutar otra tarea que esté en estado ejecutable. Este tipo de interacción puede implementarse adecuadamente como se analizó en los capítulos anteriores.

- Las variables compartidas. Es un mecanismo de interacción sin sincronización. Cuando se realiza un acceso a una variable compartida, no se suspende la ejecución de la tarea que lo lleva a cabo, excepto en máquinas que proporcionan accesos de ciclo partido como vimos en el segundo capítulo. En este tipo de máquinas cuando se inicia un acceso a una variable compartida se realiza un cambio de contexto pasando el procesador a ejecutar otra tarea. Sin embargo, éste no será el comportamiento habitual y, por lo tanto, en el análisis que se expone a continuación se supondrá que el procesador queda bloqueado durante toda la transferencia. Con esta situación se plantean problemas de eficiencia cuando el procesador que ejecuta la tarea no tiene acceso directo a la memoria donde se almacena el objeto. Sin embargo, el modelo de variables compartidas de Ada presenta algunas peculiaridades que pueden mejorar esta situación.

4.2.1 Variables compartidas

La existencia de variables compartidas entre las tareas es, en parte, consecuencia de la ortogonalidad en las reglas de ámbito (*scope*) de Ada: Una unidad, y por lo tanto una tarea, puede referenciar las variables declaradas en una unidad que la englobe.

Siguiendo la clasificación expuesta en [FLY89], existen dos tipos de variables compartidas en Ada: Síncronas y asíncronas.

- *Variables Compartidas Síncronas*: Las variables compartidas de Ada son implícitamente de este tipo. El programador debe asegurar el acceso controlado, entre puntos de sincronización, a las mismas. Para ello se deben cumplir las siguientes condiciones:

  - Si una tarea actualiza una variable ninguna otra tarea debe leer o actualizar dicha variable.

  - Si una tarea lee una variable ninguna otra tarea debe actualizar dicha variable.

  En resumen, se permiten múltiples lectores o un escritor exclusivo (acceso CREW: *Concurrent Readers Exclusive Writer*).
  Las tareas usan este tipo de variables para comunicarse información, utilizando algún mecanismo de sincronización para asegurar el acceso controlado.

- *Variables Compartidas Asíncronas*: En este grupo se incluyen las variables afectadas por el *pragma* SHARED. Este pragma sólo puede aplicarse a variables de tipo escalar o acceso y, una determinada implementación debe restringir el uso de este pragma a objetos cuya lectura o actualización constituye una operación indivisible. El acceso a este tipo de variables puede ser indiscriminado. Se utilizan, normalmente, para establecer mecanismos de sincronización (p.ej. *locks*) basados en variables compartidas. También se pueden utilizar en aplicaciones en las que es aceptable para una tarea obtener valores parcialmente actualizados [BUR85].

Ada permite que una implementación pueda mantener copias locales de las variables compartidas. Estas copias locales deben ser actualizadas en unos determinados puntos de sincronización. Para variables síncronas los puntos de sincronización son el inicio y fin de una cita, así como la activación de una tarea y su transición al estado *completada*. En el caso de variables asíncronas toda lectura y escritura debe ser un punto de sincronización.

La utilización de variables compartidas no es la única forma de que las tareas compartan datos. Una tarea puede recibir como parámetro de una cita un valor del tipo acceso con lo que tanto la tarea que llama como la llamada tendrán acceso al mismo objeto almacenado en el *heap*. El MRL no especifica si pueden mantenerse copias locales de estos objetos ni si debe existir un acceso controlado a los mismos. Por lo tanto no se incluirá este tipo de compartición de datos en nuestro análisis.

Un programa que produzca diferentes resultados, dependiendo de que la implementación mantenga copias locales o no, será erróneo. El programador olvida muchas veces este tema y supone implícitamente que todas las tareas "ven", en todo momento, el valor actualizado de las variables compartidas.

Por ejemplo, el programa propuesto en [BEN90] puede funcionar incorrectamente si la implementación mantiene copias locales de las variables compartidas. En este artículo se proponen soluciones a algunos de los problemas relacionados con el tratamiento de interrupciones en un Sistema de Tiempo Real. En estos sistemas, normalmente, la respuesta a una interrupción se divide en dos fases:

- Un manejador de interrupciones (*Interrupt_handler*) que responde a la petición de interrupción y realiza el tratamiento más crítico en tiempo.

- Una tarea de interrupciones (*Interrupt_task*) que lleva a cabo las operaciones menos críticas en tiempo asociadas a la interrupción.

El problema es de qué manera el manejador avisa a la tarea de interrupciones de la llegada de una interrupción sin quedarse bloqueado.

La solución propuesta en el artículo es la siguiente:

```
Task_Ready: Boolean:= False;
Interrupt_Occurred: Boolean:= False;
task body Interrupt_Handler is
   begin
      loop
         accept Interrupt do
            -- Muestrear datos
            if not Task_Ready then
               Interrupt_Occurred := True;
```

```
            else
                Interrupt_Task.Signal;
            end if;
        end Interrupt;
    end loop;
end Interrupt_Handler;
task body Interrupt_Task is
begin
    loop
        Task_Ready := True;
        if not interrupt_Occurred then
            accept Signal;
        else
        Interrupt_Ocurred := False;
    end if;
    Task_Ready := False;
    --Procesar datos
    end loop;
end Interrupt_Task;
```

En esta solución se utilizan las variables globales *Task_Ready* e *Interrupt_Ocurred* para controlar, respectivamente, si la tarea de interrupción está lista para aceptar la llamada del manejador, y si existe alguna interrupción que ha sido tratada por el manejador pero está pendiente de ser procesada por la tarea de interrupción. Estas dos variables están declaradas implícitamente como síncronas por lo que la implementación puede mantener copias locales de ambas variables. En ese caso, el siguiente escenario puede ocurrir:

- Situación inicial. Las tareas acaban de realizar una cita (*Signal*) y, por lo tanto, sus copias locales están actualizadas: Las copias locales de *Task_Ready* tienen el valor *true* y las copias locales de *Interrupt_Ocurred* tienen el valor *false*.

- *Interrupt_Task* pone a *false* su copia local de *Task_Ready* y comienza a procesar los datos.

- *Interrupt_Handler* acepta una interrupción (cita *Interrupt*), muestrea los datos y comprueba el valor de *Task_Ready*. Pero como *Interrupt_Task* no ha pasado por ningún punto de sincronización desde la última cita *Signal*, *Interrupt_handler* "ve" un valor *true* en *Task_Ready* y se queda bloqueada llamando al punto de entrada *Signal*.

Esta situación, en la que el manejador se queda esperando por la tarea de interrupción, era precisamente lo que se quería evitar con la solución propuesta.

Como se puede observar en el escenario descrito, las tareas no respetan un acceso CREW entre puntos de sincronización y por lo tanto el programa es erróneo. Estas variables deben estar declaradas con el pragma SHARED si se desea que su valor esté continuamente actualizado.

Pero, aún así, la solución propuesta es errónea ya que parte del supuesto de que las instrucciones del manejador de interrupciones, más concretamente las instrucciones incluidas en la cita *Interrupt*, se ejecutan como una sección crítica. Esta suposición está apoyada en las reglas de prioridad que define Ada para la aceptación de interrupciones: La cita de aceptación de una interrupción se ejecuta con una prioridad mayor que cualquier tarea definida por el usuario. Pero, como se expone en [ICH86], las prioridades son un mecanismo para indicar grados de urgencia relativos y no deben utilizarse como una técnica para obtener exclusión mutua. Si existiesen dos procesadores, por ejemplo, la tarea *Interrupt_Task* puede estar ejecutándose mientras *Interrupt_Handler* acepta la interrupción pudiendo dar lugar a la siguiente situación:

- Situación inicial. Las variables *Task_Ready* e *Interrupt_Ocurred* tienen valor de *false*.

- *Interrupt_Handler* comprueba el valor de *Task_Ready* (if not *Task_Ready*...)

- *Interrupt_Task* pone a *true* la variable *Task_Ready* y comprueba el valor de *Interrupt_Ocurred* (**if not** *Interrupt_Ocurred*...) quedándose bloqueada en el **accept** del punto de entrada *Signal*.

- *Interrupt_Handler* pone a *true* la variable *Interrupt_Ocurred*.

La tarea *Interrupt_Task* se queda bloqueada cuando está pendiente de procesar una interrupción.

En [BRY90] se presentan varios ejemplos que muestran el uso correcto e incorrecto de variables compartidas.

La posibilidad de mantener copias locales de variables globales permite que una implementación optimice el acceso a las variables compartida manteniendo copias locales en registros [BUR85] o en memoria cache [FLY89]. Además, "relaja" el modelo de memoria compartida de Ada facilitando su implementación en máquinas sin memoria compartida [FIS86] [MUN86]. La granularidad de un programa Ada, gracias a las copias locales, será más gruesa ya que sólo se realizarán accesos a los datos compartidos en los puntos de sincronización.

4.2.1.1 Gestión estática de copias locales

Se analiza, a continuación, la gestión de las copias locales. En dicho análisis se va a observar que, a pesar de las facilidades que proporcionan las copias locales, su implementación lleva asociados algunos problemas.

Para realizar este análisis se supondrá, en principio, que existe una copia global que será actualizada en los puntos de sincronización. Posteriormente se estudiarán las repercusiones de la no existencia de una copia global.

En una primera aproximación [BRY90] la gestión de las copias locales podría ser la siguiente:

- Cuando una tarea llega a un punto de sincronización se deben actualizar las copias globales de todas las variables compartidas que dicha tarea ha "escrito" desde el anterior punto de sincronización.

- En el punto de sincronización debe actualizarse el valor de las copias locales de dicha tarea. Realmente sólo se necesita actualizar las copias locales que dicha tarea va a leer hasta el próximo punto de sincronización.

Veamos un ejemplo:

```
A,B,C,D: .....;
task T1;
```

```
task T2 is
   entry E;
end T2;
task body T1 is
begin
   .....
   T2.E; (1)
   .....
   A:= ..B..;
   .....
   T2.E;(2)
   .....
   D:= ..C..;
end T1; (3)
task body T2 is
begin
   .....
   accept E; (1)
   .....
   C:= ..D..;
   .....
   accept E; (2)
   .....
   B:= ..A..;
   ....
end T2; (3)
```

Este fragmento es correcto ya que las tareas realizan un acceso CREW.

Si X_GLOBAL es la copia global de la variable X y X_LOCAL_I es la copia local de X en la tarea Ti, en los puntos señalados se producirán las siguientes acciones:

(1) B_LOCAL_1:= B_GLOBAL y D_LOCAL_2:= D_GLOBAL

(2) A_GLOBAL:= A_LOCAL_1 ; C_GLOBAL:= C_LOCAL_2; C_LOCAL_1:= C_GLOBAL y A_LOCAL_2:= A_GLOBAL

(3) D_GLOBAL:= D_LOCAL_1 Y B_GLOBAL:= B_LOCAL_2

Los puntos (1) y (2) se corresponderían con el comienzo de las citas correspondientes, y (3) con el paso al estado de completadas de ambas tareas. Existen también puntos de sincronización al final de cada cita, pero no se han incluido ya que las citas no tienen cuerpo de sentencias asociado y, por lo tanto, no existen accesos a variables compartidas por lo que no serían necesarias las actualizaciones. Tampoco se

han incluido los puntos de sincronización correspondientes a la activación de ambas tareas.

Es interesante resaltar que la gestión de la actualización debe ser hecha de tal forma que, cuando dos tareas llegan a un punto de sincronización, se pueda asegurar que una tarea no actualiza su copia local de la variable antes de que la otra haya actualizado la copia global de dicha variable. En el punto (2) del ejemplo hay que asegurar que las acciones de actualizar A_GLOBAL y C_GLOBAL preceden a las actualizaciones de A_LOCAL_2 y C_LOCAL_1.

Problemas con las sentencias condicionales

En el ejemplo propuesto, el compilador, a partir del análisis del texto del programa, puede incluir en los lugares adecuados las primitivas que se encarguen de las actualizaciones de las copias locales. El problema es que no siempre se puede conocer estáticamente las variables que deben ser actualizadas un determinado punto de sincronización.

Veamos un ejemplo:

```
A: ....;
task T1;
task T2 is
   entry E;
end T2;
task body T1 is
begin
   for I in 1..10 loop
      A:= F(A);
      If (I mod 3 = 0) then
         T2.E; (1)
      end if;
      T2.E; (2)
   end loop;
end T1;
task body T2 is
begin
   for I in 1..10 loop
      If (I mod 3 = 0) then
         accept E; (1)
         A:= G(A);
      end if;
      accept E; (2)
```

```
    end loop;
end T2;
```

El fragmento de programa es correcto ya que los accesos de las tareas responden a una política CREW. La tarea T1 accede a A cuando I no es múltiplo de 3, mientras que T2 accede cuando I es múltiplo de 3 (por la construcción del programa los índices avanzan síncronamente).

En este ejemplo no se puede determinar en tiempo de compilación qué variables deben actualizarse en cada punto de sincronización:

- Cuando la tarea T2 alcanza el punto de sincronización correspondiente a la cita (2) existen dos posibles situaciones:
  - I es múltiplo de 3. El anterior punto de sincronización se corresponde con la cita (1). Desde entonces T2 ha accedido a la variable A por lo que debe actualizar la copia global de la misma.
  - I no es múltiplo de 3. El anterior punto de sincronización corresponde con la cita (2) (ambos puntos de sincronización corresponden a la misma sentencia del programa). Desde ese punto T2 no ha accedido a la variable A por lo que no debe actualizar la copia global.
- Cuando la tarea T1 alcanza el punto de sincronización correspondiente a la cita (2) existen dos posibles situaciones:
  - I es múltiplo de 3. El anterior punto de sincronización se corresponde con la cita (1). Desde entonces la tarea no ha accedido a A por lo que no debe actualizar la copia global.
  - I no es múltiplo de 3. El anterior punto de sincronización se corresponde con la cita (2) (ambos puntos de sincronización se corresponden con la misma sentencia del programa). Desde ese punto T1 ha accedido a A, por lo que debe actualizar su copia global.

La secuencia de sincronización que va ejecutando el programa, así como las variables accedidas entre dos puntos de sincronización, sólo pueden ser determinadas en tiempo de ejecución.

Para mantener una solución estática (en tiempo de compilación) se deberían añadir nuevos puntos de sincronización en las sentencias del lenguaje que impliquen caminos alternativos en la ejecución (**if, case, exit, for,** ...). Esta solución podría generar un gran número de actualizaciones innecesarias que degradan el rendimiento pudiéndose, incluso, aproximar a una solución sin copias locales.

Problemas con los subprogramas

Existen también dificultades en la gestión de copias locales cuando los puntos de sincronización están incluidos en subprogramas.

Veamos un ejemplo:

```
A,B,C,D,E: ...;
task CONTROL is
   entry ENTRAR;
   entry SALIR;
end CONTROL;
task body CONTROL is
begin
   loop
      accept ENTRAR;
      accept SALIR;
   end loop;
end CONTROL;
procedure OBTENER_ACCESO is
begin
   CONTROL.ENTRAR;
   -- Acceso a D
   .....
end OBTENER_ACCESO;
procedure LIBERAR_ACCESO is
begin
   .....
   -- Acceso a E
   CONTROL.SALIR;
end LIBERAR_ACCESO;
task T1;
task T2;
task T3;
```

```
task body T1 is
begin
   loop
      OBTENER_ACCESO;
      -- Acceso a A y B
      LIBERAR_ACCESO;
   end loop;
end T1;
task body T2 is
begin
   loop
      OBTENER_ACCESO;
      -- Acceso a B y C
      LIBERAR_ACCESO;
   end loop;
end T2;
task body T3 is
   begin
   loop
      OBTENER_ACCESO;
      -- Acceso a A y C
      LIBERAR_ACCESO,
   end loop
end T3;
```

El programa es correcto ya que las tareas siguen una política CREW en sus accesos. En este ejemplo los puntos de sincronización de las tareas T1, T2 y T3, excluyendo su activación y terminación, no están dentro del cuerpo de la propia tarea sino en el contexto de procedimientos invocados por dichas tareas.

Cuando una tarea invoca el procedimiento LIBERAR_ACCESO y se alcanza el punto de sincronización correspondiente a la cita SALIR se deben actualizar las copias globales de las variables que han sido actualizadas desde el último punto de sincronización. Si la tarea que llama es T1 se actualizarán las copias globales de A, B, D y E; si es T2 se actualizarán las de B, C, D y E; y si se trata de T3 se actualizarán las de A, C, D y E. Por lo tanto, no puede conocerse en tiempo de compilación qué copias globales deben ser actualizadas en el primer punto de sincronización de un subprograma, ya que depende de la tarea que lo haya invocado.

Las llamadas a subprogramas, cuando la rutina invocada incluya algún punto de sincronización, deben ser, en una solución estática, un nuevo punto de sincronización de las copias locales. Así, en el primer punto de sincronización que se ejecute

dentro del subprograma, ya sea una llamada a un punto de entrada o la activación de una tarea, solamente se sincronizarán las copias locales de las variables compartidas actualizadas desde la activación del subprograma hasta ese instante, con independencia de la tarea que invocó la rutina. En el ejemplo, la tarea T1 antes de invocar el procedimiento LIBERAR_ACCESO debe actualizar las copias globales de A, B; T2 las de B y C; y T3 las de A y C. Esta sincronización debe realizarse siempre que el subprograma invocado incluye algún punto de sincronización, o bien, que, sin incluir ningún punto de sincronización, pueda invocar a algún subprograma que contenga algún punto de sincronización. Siguiendo con el ejemplo, cuando dentro del contexto del procedimiento LIBERAR_ACCESO, independientemente de la tarea que lo llamó, se alcanza el punto de entrada SALIR sólo se actualizarán las copias globales de D y E.

Las llamadas a subprogramas desde otros subprogramas presentan problemas similares a las llamadas a subprogramas incluidas directamente en el código de una tarea y, por lo tanto, se les aplica el mismo tratamiento.

Veamos un ejemplo:

```
A, B, C: ....;
procedure P is
begin
   ....
   Actualización de C
   T.E; (llamada a un punto de entrada)
   ....
end P;
procedure Q is
begin
   ....
   Actualización de B
   P;
   ....
end Q;
task T;
task body T is
begin
   ....
   Actualización de A
   Q;
   ....
end T;
```

En este ejemplo, las sincronizaciones, con una solución estática, tendrán lugar en los siguientes instantes:

- Antes de que T invoque el procedimiento Q se debe actualizar la copia global de A, ya que Q, aunque no incluye ningún punto de sincronización, invoca el procedimiento P que incluye una llamada a un punto de entrada.

- Antes que Q llame a P se debe actualizar la copia global de B puesto que P incluye un punto de sincronización. Si no se realiza esta sincronización no se podría conocer en tiempo de compilación, al igual que en el caso de las llamadas directas desde la tarea, que copias locales deben sincronizarse.

- En la llamada al punto de entrada E debe actualizarse la copia global de C.

Otro problema relacionado con la gestión de copias locales y subprogramas es el acceso desde un subprograma a las copias locales de la tarea que lo invoca. Del análisis del programa se conocen las variables compartidas que referencia, directa o indirectamente, cada tarea. En el último ejemplo la tarea T accede a las variables compartidas A (directamente), B y C (ambas indirectamente). Las referencias a variables compartidas desde un subprograma deben ser sustituidas por referencias a las copias locales asociadas a la tarea que invoca el subprograma. La determinación del lugar donde se almacena la copia local puede ser problemático sin apoyo hardware, sobre todo cuando el nivel de anidamiento del subprograma es menor que el de la tarea. Una alternativa es crear las copias locales de las variables compartidas accedidas por un subprograma dentro del propio subprograma como si fueran variables locales del mismo. El ejemplo anterior, bajo el prisma de la primera opción (copias locales asociadas a la tarea), podría interpretarse de la siguiente forma:

```
A, B, C ....;
procedure P is
begin
   ....
   actualización de C_LOCAL
   T.E;
end P;
procedure Q is
```

```
begin
    ....
    actualización de B_LOCAL
    P;
    ....
end Q;
task T;
task body T is
    A_LOCAL, B_LOCAL, C_LOCAL: ...
begin
    ....
        actualización de A_LOCAL
    Q;
    ....
end T;
```

Bajo la segunda opción (copias locales asociadas a subprogramas) la interpretación podría ser:

```
A, B, C: ....;
procedure P is
    C_LOCAL: ....;
begin
    ....
    actualización de C_LOCAL
    T.E;
    ....
    end P;
procedure Q is
    B_LOCAL: ....;
begin
    ....
    actualización de B_LOCAL
    P;
    ....
end Q
task T;
task body T is
    A_LOCAL: ....;
begin
    ....
    actualización de A_LOCAL
    Q;
    ....
end T;
```

Hay que resaltar que ambos fragmentos de programa sólo son un intento de facilitar la comprensión de las alternativas expuestas. No se trata de una posible implementación de las mismas. La utilización de la segunda opción simplifica las referencias a las copias locales desde un subprograma pero presenta algunas dificulta-

des. Esta opción hace posible la existencia de múltiples copias locales de una misma variable compartida. Si una tarea accede a una variable compartida y, posteriormente, invoca a un subprograma que también accede a dicha variable, coexistirán dos copias locales de la variable compartida asociadas, directa o indirectamente, a la tarea. Para evitar problemas de coherencia entre ambas copias cada llamada a subprograma debe ser un punto de sincronización, al igual que en la solución estática, en el que se actualicen las copias globales de las variables que han sido escritas por la tarea (o subprograma llamante) desde el último punto de sincronización y se asigne valor inicial, a partir de las copias globales, a las copias locales del subprograma. Asimismo, al finalizar el subprograma debe realizarse el proceso inverso: Actualizar las copias globales a partir de las copias locales del subprograma, y actualizar las copias locales de la unidad que invocó el subprograma.

Del análisis se desprende que la gestión en tiempo de compilación de las copias locales de variables compartidas implica la inclusión de nuevos puntos de sincronización, además de los señalados en la definición del lenguaje. Estos nuevos puntos de sincronización pueden llevar a aumentar considerablemente el número de actualizaciones y, por lo tanto, disminuir la efectividad del mecanismo de copias locales. En opinión del autor el análisis de algoritmos que, a partir del análisis estático del programa minimicen el número de sincronizaciones puede ser un campo de estudio interesante.

4.2.2 Gestión dinámica de copias locales

El problema de las soluciones estáticas es que no se conoce la suficiente información, en tiempo de compilación, para decidir las variables que deben ser sincronizadas. Frente a este tipo de solución, existen alternativas de carácter dinámico, en las que se decide en tiempo de ejecución qué variables deben ser sincronizadas, teniendo en cuenta el estado de las copias locales. A continuación se propone un algoritmo dinámico (A_1) para la gestión de las copias locales. El algoritmo está basado en las siguientes consideraciones:

- Cuando se alcanza un punto de sincronización se actualizan las copias globales de las variables compartidas que han sido escritas desde el anterior punto de sincronización. Para ello es necesario guardar, junto con la copia local, información que determine si dicha copia ha sido escrita desde el anterior punto de sincronización.

- Las actualizaciones de las copias locales no tienen lugar en los puntos de sincronización sino cuando la tarea realiza un acceso de lectura a la variable correspondiente. Además, dichas actualizaciones no tendrán lugar si en el intervalo de tiempo entre los dos últimos puntos de sincronización, la tarea accedió (lectura o escritura) a dicha variable, puesto que esto implica que durante dicho intervalo ninguna otra tarea accedió a la variable (acceso CREW) y, por lo tanto, la copia local está actualizada. Es necesario almacenar con cada copia local información que determine si dicha copia ha sido accedida en dicho intervalo.

Una descripción informal del algoritmo sería:

- Información que se almacena con cada copia:

  - *actualizada.* Información de carácter booleano que indica si la copia local ha sido escrita desde el anterior punto de sincronización. Inicialmente *falsa.*

  - *válida.* Puede tomar tres valores:

    - *Cero.* La variable no ha sido accedida en el intervalo entre los anteriores puntos de sincronización. Por lo tanto, para el algoritmo propuesto la copia es inválida.

    - *Uno.* La variable ha sido accedida en el último intervalo.

    - *Dos.* La variable ha sido accedida desde el último punto de sincronización (actual intervalo).

  Inicialmente vale cero.

- Los instantes significativos para el algoritmo son:

- Punto de sincronización. Se realizarán para cada copia local las siguientes acciones:
  - **Si** *actualizada* **entonces** Actualizar la copia global correspondiente y desactivar *actualizada.*
  - Decrementar *válida* (si era cero seguirá siéndolo).
- Lectura. Por cada copia local se llevan a cabo las operaciones siguientes:
  - **Si** *válida*= 0 **entonces** actualizar la copia local a partir de la copia global correspondiente.
  - Poner *válida* igual a 2.
- Escritura. Por cada copia local se realizan las operaciones siguientes:
  - Activar *actualizada.*
  - Poner *válida* igual a 2.

Con el algoritmo propuesto una copia puede ser considerada inválida aunque no lo sea, ya que el hecho de que una tarea no haya accedido en el intervalo entre los dos últimos puntos de sincronización a una variable no implica que haya existido otra tarea que cambie el valor de la misma. Esta deficiencia es debida a que el algoritmo basa sus decisiones en información local a la tarea. Un algoritmo que generase únicamente las actualizaciones necesarias debería basarse en información más global, lo que implica una mayor complejidad en la implementación del mismo. Un posible algoritmo basado en información global sería el siguiente:

- Información que se almacena en cada copia.
  - *actualizada.* Tiene el mismo significado que en el anterior algoritmo.
  - *válida.* Información de carácter booleano que indica si la copia es válida.

- los instantes significativos del algoritmo son:

  - Punto de sincronización. Las operaciones a realizar por cada copia local son:

    - **Si** *actualizada* **entonces** Actualizar la copia global correspondiente, desactivar *actualizada* e invalidar las copias locales correspondientes a dicha variable que existan en otras tareas (*válida:=false*).

  - Lectura. Si la copia local está invalidada actualizarla y activar *válida.*

  - Escritura. Activar *actualizada.*

Este algoritmo, aunque genera únicamente las actualizaciones necesarias, es mucho más difícil de implementar que el anterior, sobre todo el mecanismo de invalidación desde una tarea de copias locales de otras tareas. Además va a generar muchas invalidaciones innecesarias: Una copia local de una tarea puede recibir un gran número de invalidaciones antes de que la tarea acceda a dicha copia.

Otra de las características de los algoritmos expuestos es que son independientes de las peculiaridades de cada programa. Se pueden plantear algoritmos que aprovechen estas peculiaridades (por ejemplo relaciones de precedencia entre tareas) del programa correspondiente para facilitar la gestión de las copias locales.

Los algoritmos dinámicos permiten conocer las variables compartidas que deben actualizarse al llegar un punto de sincronización. Sin embargo, no resuelven el problema de acceder desde un subprograma a las copias locales de la tarea que lo invoca. Esto significa que las copias locales estarán asociadas a los subprogramas y tanto las llamadas como los retornos de subprogramas deben seguir siendo puntos de sincronización.

Una posible solución a este problema y, en general, una manera de acelerar toda la gestión de las copias locales es utilizar una memoria tipo cache para realizar esta gestión.

Utilización de memorias cache para la gestión de copias locales

Se necesita un mecanismo del tipo memoria cache que gestione "automáticamente" el estado de las copias locales. El problema clave cuando se utilizan memorias cache en máquinas con múltiples procesadores es mantener la coherencia entre las múltiples copias que pueden existir en las diversas caches. La solución tradicional a este problema es la inclusión de protocolos hardware (p.ej. protocolos *snoopy*) que se ocupen de mantener la coherencia. Estos protocolos implican un hardware complejo y presentan problemas cuando aumenta el número de procesadores del sistema. Frente a esta alternativa surgen los esquemas de coherencia software o dirigidos por el compilador [STE90]. Estos esquemas mantienen la coherencia de las caches a partir del análisis del programa correspondiente. La clave de los esquemas software [CHE90] es "relajar" el requerimiento de consistencia entre las caches: Un dato almacenado en una cache de un procesador puede no estar actualizado, con tal de que dicho procesador no use este dato.

El compilador puede detectar intervalos en los que un procesador puede mantener una copia local de una variable en su cache ya que, por la construcción del programa, ningún otro procesador intentará actualizar dicha variable. Según se van atravesando estos intervalos el procesador ejecutará instrucciones que actualizan la copia en memoria de algunas variables (no sería necesario con una política *write-through*) y otras instrucciones que invaliden algunas copias en la cache.

Los esquemas de coherencia software requieren un hardware más sencillo y, generalmente, conducen a soluciones en las que cada procesador gestiona su propia cache sin necesidad de comunicación entre los procesadores.

Estos esquemas software son muy apropiados para el modelo de datos Ada [FLY89]. El comportamiento de las variables compartidas síncronas de Ada responden perfectamente a estos esquemas, por lo tanto, se podrán mantener copias de dichas variables en las diferentes caches generándose actualizaciones e invalidaciones en los puntos de sincronización. Por otra parte, no se deben mantener copias locales

de las variables compartidas asíncronas en las caches ya que deben estar continuamente actualizadas. El compilador, por lo tanto, marcará este tipo de variables como "no cacheables".

Una versión del algoritmo de sincronización A_1 podría ser utilizada para la gestión de la coherencia de las caches. Las caches, suponiendo una política *write-through*, deberían estar diseñadas de la siguiente forma:

- Junto con cada dato en la cache se guardan dos bits:
  - *válido*. El dato almacenado es válido.
  - *accedido*. Ha existido, desde el último punto de sincronización, un acceso (lectura o escritura) a ese dato.
- La cache debería soportar las siguientes operaciones (que serán invocadas en cada punto de sincronización):
  - Invalidar todas las copias que no tengan activo el bit *accedido* y, posteriormente, desactivar dicho bit para todas las copias.
- En las lecturas y escrituras la cache debe llevar a cabo las siguientes acciones:
  - Toda lectura y escritura debe activar el bit *accedido* de la copia correspondiente.
  - Cuando se produce un fallo en la cache (no existe la copia o es inválida) se lee la copia global, se actualiza la cache y se activa el bit *válido* de la copia correspondiente.

Se ha supuesto una política de actualización *write-through* que mantiene la copia en la memoria siempre actualizada. Una alternativa es actualizar las copias en memoria sólo en los puntos de sincronización con lo que se disminuyen los accesos a memoria. En este caso, junto con la copia hay que almacenar un bit de *cambio* que se activará cuando se haya modificado el valor de la copia. La cache debería propor-

cionar una operación *Actualizar* (invocada en los puntos de sincronización) que actualizara las copias en memoria correspondientes a todas las copias con el bit de *cambio* activo y posteriormente "reseteara" dicho bit.

En el algoritmo se ha supuesto también que cada tarea está asociada a un único y exclusivo procesador durante su existencia (correspondencia **m=n** con asignación estática).

Si existen varias tareas asociadas al mismo procesador (**m>n**), dicho procesador ejecutará a lo largo del tiempo el código asociado a distintas tareas y, por lo tanto, en su cache se almacenarán copias locales pertenecientes a diferentes tareas. Cuando se produce un punto de sincronización en la ejecución de una tarea se invalidarán todas las copias con el bit *accedido* igual a *false*, independientemente de si están asociadas a la tarea que alcanzó el punto de sincronización. Se generarán. por lo tanto, muchas invalidaciones innecesarias.

Para evitar esta situación habría que asociar con cada copia un identificador que especifique a qué tarea está asociada, de forma que la operación *invalidar* se lleve a cabo únicamente sobre las copias de la tarea que llegó al punto de sincronización. Esta solución, además de necesitar una cache más compleja, puede presentar problemas cuando existe una copia asociada a varias tareas ya que se necesitaría almacenar el identificador de cada de una de las tareas.

Una alternativa sería tener una copia por cada tarea aunque se correspondan todas las copias con la misma variable compartida. Esta alternativa implica que el direccionamiento de la cache se debe realizar no sólo con la dirección de la variable global, sino también con el identificador de la tarea que realiza el acceso.

Si existe asignación dinámica, la tarea, durante su existencia, puede ser ejecutada por diferentes procesadores. Es necesario, por lo tanto, que cuando una tarea "migre" de procesador las copias almacenadas en la cache del procesador origen sean invalidadas. Es importante resaltar que la "migración" de tareas no suele realizarse

entre procesadores que no comparten memoria, ya que la transferencia entre las memorias del código, los datos y el estado de la tarea será bastante ineficiente.

La utilización de memorias cache resuelve otro de los problemas expuestos anteriormente: El direccionamiento desde un procedimiento de las copias locales de la tarea que lo invoca (directa o indirectamente). Al utilizar caches todas las referencias a las variables compartidas se transforman, por hardware, en referencias a las copias almacenadas en la cache. Por lo tanto, utilizando caches no es necesario que las llamadas a subprogramas sean puntos de sincronización de las copias locales.

Sin embargo, no siempre interesa que un subprograma invocado por una tarea utilice las copias locales asociadas a la tarea (almacenadas en la cache del procesador que ejecuta la tarea). Esto es debido a que, en máquinas sin memoria compartida, la tarea y el subprograma invocado pueden estar asociados a distintos procesadores y, por lo tanto, a distintas caches. En este caso sería necesario realizar una sincronización de las copias en la invocación y retorno del subprograma, como en la solución estática.

En máquinas con memoria compartida el procesador que ejecuta la tarea accederá directamente al código del subprograma. En este caso, si que sería interesante que desde el subprograma se accediese a las copias de la tarea.

4.2.2.1 Gestión de copias locales sin una copia global

Toda la exposición realizada hasta ahora se ha basado en la existencia de una copia global de cada variable y múltiples copias globales para acelerar los accesos a la copia global. En [BRY90] se plantea la implementación del mecanismo de copias locales sin la existencia de una copia global de la variable. La estrategia seguida es almacenar junto con cada copia local un *time-stamp* que permita establecer, de alguna forma, el orden parcial en los accesos a la variable compartida correspondiente (el autor no da más detalles de la gestión del *time-stamp*). Cuando dos tareas se sincronizan entre sí, intercambian sus copias locales seleccionando la copia cuyo *time-stamp* indica que ha sido accedida más recientemente.

Esta solución, por lo tanto, sustituye la sincronización entre copias locales y copias globales que se llevaba a cabo en la solución con copia global, por el intercambio de copias locales con *time-stamp* entre las tareas que se sincronizan. En nuestra opinión esta estrategia puede ser problemática.

En la solución con copia global cuando una tarea alcanza un punto de sincronización se actualizan las copias globales correspondientes, las cuales quedan disponibles para el acceso controlado por parte de otras tareas.

En la solución sin copia global, después de una sincronización sólo quedan actualizadas las copias locales de las tareas implicadas en la misma.

Los problemas pueden surgir cuando hay más de dos tareas que acceden de forma controlada a variables compartidas. Supongamos que existen tres tareas (Ti, Tj, Tk) accediendo a variables compartidas siguiendo la siguiente secuencia de accesos:

- Ti actualiza un subconjunto de variables compartidas y, a continuación, se sincroniza con Tj pasándole el derecho de acceso.

- Tj actualiza, después de la sincronización con Ti, un subconjunto de variables compartidas (algunas serán comunes con las de Ti y otras no) y, a continuación, se sincroniza con Tk pasándole el derecho de acceso.

- Tk actualiza, después de la sincronización con Tj, un subconjunto de variables compartidas (algunas serán comunes con las de Ti, otras con las de Tj y otras no comunes).

Supongamos también que existe una variable X que Ti y Tk actualizan pero Tj no.

Con estos supuestos, existirán sendas copias locales de X asociadas a Ti y Tk. Sin embargo, no existirá una copia de X asociada a Tj, ya que esta tarea no accede a X. De esta forma cuando Ti y Tj se sincronicen, Tj no obtendrá el valor actualizado de X y, por lo tanto, no se le podrá proporcionar a Tk cuando se sincronice con ella.

El problema surge debido a que Tk no recibe el derecho de acceso a X directamente desde Ti, sino a través de Tj, por lo que no existe una sincronización directa entre ambas tareas. Este problema no ocurriría si existiese una copia global, ya que en ese caso, Tk accedería a la copia global de X que habría sido convenientemente actualizada cuando se sincronizaron Ti y Tj.

Para poder utilizar la solución propuesta en [BRY90], una tarea debería almacenar no solamente las copias locales de las variables que accede, sino cualquier copia local que recibe de otra tarea cuando se produce una sincronización entre ambas. Es importante resaltar que esto puede implicar que una tarea mantenga copias locales de variables que están fuera de su ámbito. En el siguiente programa se puede ver un ejemplo de esta situación.

```
procedure P is
   A,B:...;
   task T1 is
      entry E1;
      entry E2;
   end T1;
   task body T1 is
   begin
      accept E1;
      Actualizar A y B
      accept E2;
   end T1;
begin
   declare
      C:...;
      task T2;
      task T3;
      task body T2 is
      begin
         Actualizar A y C
         T1.E1;
      end T2;
      task body T3 is
      begin
         T1.E2;
         Actualizar B y C
      end T3;
   begin
      null;
   end;
end P;
```

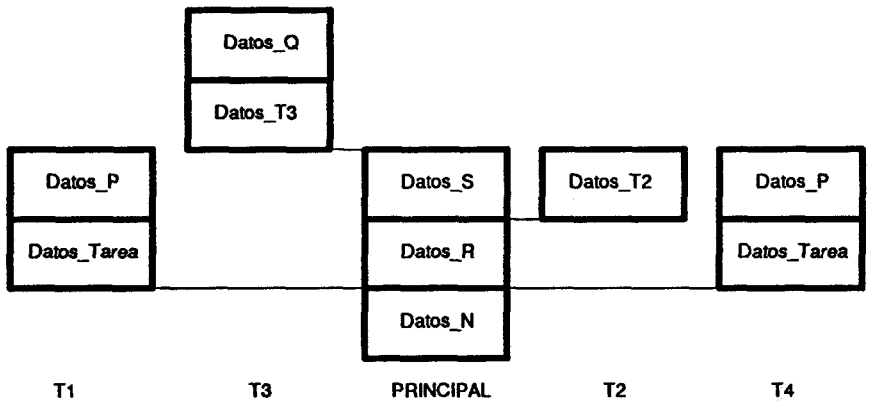

Fig. 4.1 Cactus Stack del programa

En este ejemplo, T1 debe almacenar una copia de la variable C aunque esté fuera de su ámbito, ya que, en caso contrario, no podría llegar el valor actualizado de C desde T2 a T3 a través de T1.

4.3 El anidamiento de las tareas y las reglas de ámbito

Las reglas de ámbito de Ada son análogas a las de otros lenguajes estructurados en bloques (*block structured*), pero con las implicaciones debidas a la existencia de construcciones concurrentes en el lenguaje. La pila usada para la gestión de la memoria en la mayoría de las implementaciones de lenguajes estructurados en bloques, se convierte, en el caso de Ada, en un conjunto de pilas, una por tarea, que forman lo que se suele denominar *Cactus Stack*. Así, la pila de un programa Ada está formada por varias "ramas", una por cada tarea, que crecen independientemente pero comparten el "tronco" (los niveles inferiores) del cactus. En la figura 4.1 se muestra la *Cactus Stack* asociada al programa de la figura 4.2.

El mantenimiento de estas reglas de ámbito va a implicar que se produzcan interacciones que no están incluidas explícitamente por el programador. En este apartado vamos analizar las características de estas interacciones y sus repercusiones en la eficiencia.

```
procedure PRINCIPAL is
   procedure N is
      DATOS_N:....; -- variables locales de PRINCIPAL
      task type TIPO_TAREA;
      task body TIPO_TAREA is
         DATOS_TAREA:....; -- variables locales de la tarea
         procedure P is
            DATOS_P:....; -- variables locales de P
         begin
         end P;
      begin
         P;
      end TIPO_TAREA;
      procedure Q is
         DATOS_Q:....; -- variables locales de Q
      begin
      end Q;
      procedure R is
         DATOS_R:....; -- variables locales de R
         task T2;
         task body T2 is
            DATOS_T2:....; -- variables locales de T2
         begin
         end T2;
         procedure S is
            DATOS_S:....; -- variables locales de S
            task T3;
            task body T3 is
               DATOS_T3:....; -- variables locales de T3
               T4: TIPO_TAREA;
            begin
               Q;
            end T3;
         begin
         end S;
      begin
         S;
      end R
   begin
      R;
   end N;
begin
   N;
end PRINCIPAL;
```

Fig 4.2 Programa al que corresponde el cactus stack

Dentro de un programa Ada se pueden distinguir dos tipos de variables: Variables definidas en el nivel más externo de una unidad de biblioteca y el resto de las variables.

Las referencias a objetos definidos en el nivel más externo de una unidad de biblioteca se resuelven en tiempo de compilación ya que la direcciones de dichos objetos son estáticas.

Las referencias a objetos que no estén definidos en el nivel más externo deben ser resueltas en tiempo de ejecución ya que su dirección es dinámica y depende de la traza de ejecución del programa. La referencia de una unidad a un objeto local puede resolverse sin necesidad de conocer la traza de ejecución del programa ya que su posición relativa dentro de la zona de datos locales es fija.

En un entorno en el que todos los procesadores comparten memoria existirá una estructura de datos que refleje, en cada instante, la traza de ejecución del programa. Esta estructura permitirá al procesador que esté ejecutando el código de una tarea calcular, en tiempo de ejecución, las direcciones de los objetos referenciados por dicha tarea.

Si no existe memoria compartida no se puede mantener una estructura de datos accesible directamente por todos los procesadores. Por lo tanto, debe existir algún mecanismo que permita a un procesador que esté ejecutando el código de una tarea conocer la traza de ejecución del programa.

El problema del contexto recursivo

En relación con el tema, Volz y otros [VOL89] plantean el problema del contexto recursivo. La definición del problema es la siguiente:

Supóngase que la unidad que crea un objeto (Unidad-C) y la unidad que lo referencia (Unidad-R) se ejecutan en dos procesadores que no comparten memoria. Si la unidad-C puede ser llamada recursivamente existirán simultaneamante varias instancias de la unidad-C y de los objetos declarados en la misma. Así, será necesario exportar el contexto de la unidad-C para que la unidad-R correspondiente acceda a la instancia adecuada del objeto. Veamos un ejemplo de este problema:

```
procedure P1 is -- unidad-C
   X: INTEGER;
```

```
    task T2;
    task body T2 is -- unidad-R
    begin
        .....
        X := ... -- referencia remota
        P1; -- llamada recursiva
        .....
    end T2;
begin
    .....
end P1;
```

En este ejemplo, las sucesivas llamadas recursivas a P1 crearán múltiples instancias de X y T2. Debe existir un mecanismo que proporcione a una determinada instancia de T2 la información de contexto necesaria para referenciar la instancia de X apropiada.

Como consecuencia de esta necesidad de exportar el contexto se generarán un gran número de mensajes entre los procesadores y, por lo tanto, habrá una perdida de eficiencia en la implementación. Como se verá más adelante, la solución propuesta en el artículo es restringir la manera en que se distribuye el programa Ada.

Es importante resaltar que el objeto referenciado puede ser una tarea (p.ej. una llamada a un punto de entrada). Es necesario, al igual que antes, asegurar que la unidad-R accede a la instancia adecuada de la tarea.

Veamos un ejemplo con tipos tarea anónimos:

```
procedure P is – unidad-C
    task T1 is
        entry E; – objeto referencia
    end T1;
    task T2;
    task body T1 is
        ....
    end T1;
    task body T1 is -- unidad-R
    begin
        ....
        T1.E -- referencia remota
    end T2;
begin....
    P; -- llamada recursiva
end P;
```

Las sucesivas llamadas recursivas a P crean múltiples instancias de T1 y T2. Debe existir un mecanismo que permita que una instancia de la tarea T2 llame al punto de entrada de la instancia de T1 apropiada.

En nuestra opinión el problema del contexto recursivo se engloba dentro de la problemática general de la implementación y gestión del *cactus stack* en un sistema con múltiples procesadores ya que, a partir de esta estructura, se puede calcular la dirección de la instancia apropiada del objeto referenciado. Antes de analizar dicha problemática, se expondrá la manera habitual de gestionar el *cactus stack* cuando existe memoria compartida para, a continuación, estudiar las implicaciones de la falta de memoria compartida.

Gestión del cactus stack

Los lenguajes estructurados en bloques (estilo ALGOL) utilizan usualmente *registros de activación* y *displays* para gestionar las variables de un programa. El *registro de activación* se crea cuando se activa una rutina y contiene, entre otras cosas, los parámetros y datos locales de la misma. El *display* es un vector de punteros a los registros de activación de los procedimientos más externos. Permite resolver las referencias de una rutina a datos no locales evitando la necesidad de recorrer una cadena de enlaces estáticos. La gestión del *display*, sin entrar en detalles de implementación, podría hacerse de la siguiente forma:

Un procedimiento **Q** cuyo nivel de anidamiento estático es **I** necesita **I-1** punteros en el *display* para resolver sus referencias no locales. Los **I-1** punteros necesarios se obtienen a partir de los punteros correspondientes al procedimiento **P** que invocó a **Q**. Si el nivel estático de **P** es mayor o igual que el de **Q**, se copiarán **I-1** punteros de **Q** (desde el correspondiente al nivel más externo, nivel 1, hasta I-1). Si el nivel estático de **P** es menor que el de **Q** (**P** necesariamente tiene nivel **I-1**) se copian los **I-2** punteros asociados a **P** y se añade un nuevo puntero que señale al registro de activación de **P**.

En la figura 4.3 se muestra la evolución de la pila y el *display* del programa de la figura 4.4 según se van realizando las distintas llamadas a rutinas. La figura refleja el instante en que el procedimiento P4 invoca al procedimiento P2. En ese instante el *display* activo, por lo tanto, es el correspondiente a P2 (sombreado en la figura).

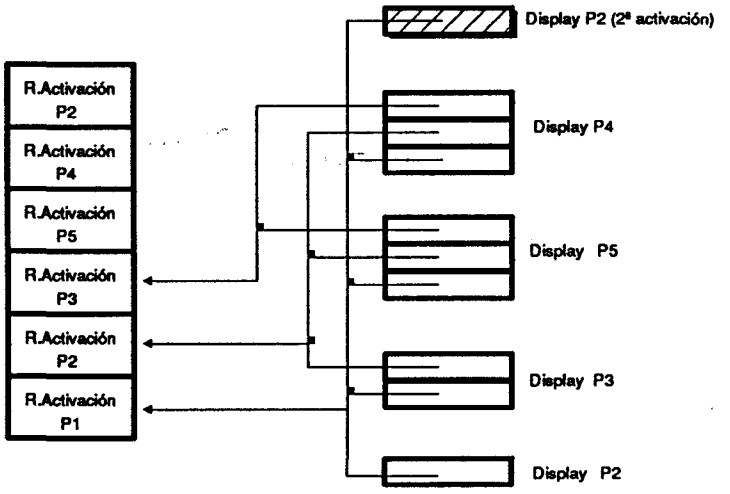

Fig. 4.3 Registro activación y display del programa

```
procedure PRINCIPAL is
   procedure P1 is
      procedure P2 is
         procedure P3 is
            procedure P4 is
            begin
               P2;
            end P4;
            procedure P5 is
            begin
               P4;
            end P5;
         begin
            P5;
         end P3;
      begin
         P3;
      end P2;
   begin
      P2;
   end P1;
begin
   P1;
end PRINCIPAL;
```

Fig. 4.4 Programa secuencial correspondiente

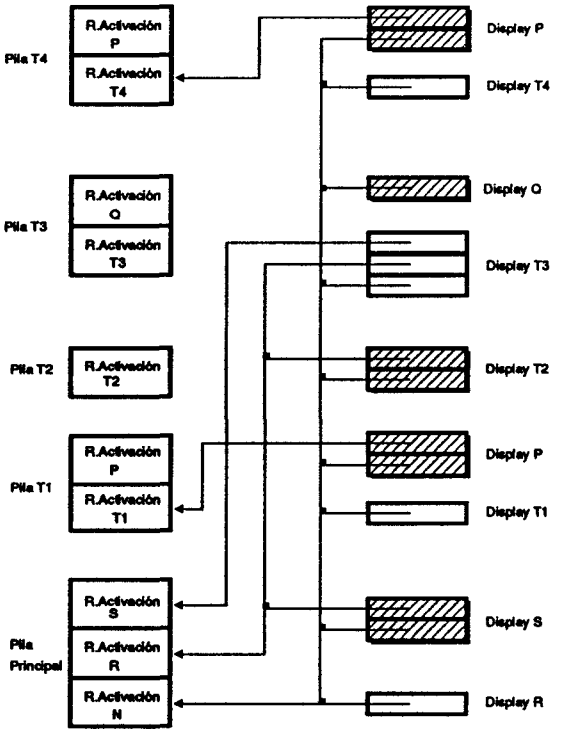

Fig. 4.5 Registros de activación y displays

La inclusión de construcciones concurrentes dentro de un lenguaje estructurado en bloques, como en el caso de Ada, complica la gestión de las variables. En vez de un único *display* debe existir uno por cada tarea. Cuando se crea una tarea con nivel de anidamiento estático **I** hay que incluir en el *display* de la misma **I-1** punteros para resolver sus referencias no locales. Como en el caso de las llamadas a rutinas se presentan dos casos:

- Si la unidad que crea la tarea tiene mayor o igual nivel se copian los **I-1** primeros punteros asociados a dicha unidad en el *display* de la nueva tarea.

- Si la unidad que crea la tarea tiene nivel **I-1** se copian los **I-2** punteros asociados a dicha unidad en el *display* de la tarea y además se incluye un puntero al registro de activación de dicha unidad.

En la figura 4.5 puede contemplarse un ejemplo de gestión de múltiples *displays* y pilas en el programa concurrente correspondiente a la figura 4.2. El instante reflejado en la figura es el siguiente:

- El flujo de control asociado al procedimiento PRINCIPAL está ejecutando el cuerpo del procedimiento S.
- La tarea T1 está ejecutando el cuerpo del procedimiento P.
- La tarea T2 está ejecutando su propio cuerpo.
- La tarea T3 está ejecutando el cuerpo del procedimiento Q.
- La tarea T4 está ejecutando el cuerpo del procedimiento P.

En la figura se muestran los *displays* activos en ese instante (sombreados en la figura). Es interesante resaltar que existe un *display* activo por cada tarea.

La estructura formada por *displays* y los registros de activación permite que las referencias desde una unidad a objetos declarados fuera de la misma se correspondan con la instancia apropiada del objeto, ya sea una tarea o cualquier otro tipo de objeto. En el ejemplo, las referencias desde el cuerpo del procedimiento S a las variables locales de R (DATOS-R) se resuelven accediendo al registro de activación de R a través del puntero almacenado en la segunda entrada del *display* de S. Por otra parte, las referencias desde el cuerpo de S a la tarea T1 se resuelven accediendo al registro de activación de N a través del puntero almacenado en la primera entrada del *display* de S, ya que en dicho registro de activación se almacena una referencia a la instancia apropiada de dicha tarea (p.ej. un puntero al bloque de control de la tarea).

Gestión en multiprocesadores

En un multiprocesador es necesario que el procesador que está ejecutando la unidad que crea una tarea o que invoca un subprograma pase la información de contexto al procesador que ejecuta la tarea o subprograma activado. Esta información estará formada por los **i-1** punteros correspondientes a una tarea o subprograma con un nivel estático **i**.

Si ambos procesadores comparten memoria pueden utilizar ésta para comunicarse la información. Cuando se trata de llamar a un subprograma, la transferencia no será necesaria ya que el procesador que realiza la llamada ejecutará también el subprograma. Si no comparten memoria el procesador "activador" mandará un mensaje al procesador "activado" con la información correspondiente.

En nuestra opinión, la gestión del *cactus stack*, ya sea en una máquina con memoria compartida o sin ella, no resulta un factor que afecte de una forma determinante en la implementación de la máquina Ada. Por un lado, las transferencias de la información de contexto no se producen con una frecuencia alta (granularidad no muy fina), sólo en las activaciones de tareas y subprogramas remotos. Por otro lado, ya eran necesarias otras transferencias de información, independientemente del *cactus stack*, en la activación de las tareas y subprogramas remotos. La información de contexto se puede incluir junto con otros datos como, por ejemplo, los parámetros de la rutina.

En las consideraciones expuestas se ha elegido un método para la gestión de memoria de un programa basado en registros de activación y *displays*. En caso de memoria compartida este método es bastante usual aunque hay otros como, por ejemplo, los *relays sets* [FLY89]. Si no existe memoria compartida el método utilizado debería tener en cuenta esto para lograr una implementación lo más eficiente posible. Sin embargo, en las consideraciones expuestas se ha extendido el uso de *displays* a estos entornos ya que, aunque no sea una solución orientada hacia los mismos, da una idea del tipo de información que deben comunicarse entre sí los procesadores.

Algunas propuestas alternativas

Existen diferentes propuestas de modificación del lenguaje Ada con respecto a los dos aspectos analizados hasta ahora en este capítulo: La interacción mediante variables compartidas y el mantenimiento del contexto.

Una posible alternativa sería prohibir que las tareas utilicen variables compartidas para comunicarse dejando la cita como único mecanismo de interacción, y mantener las reglas de ámbito de Ada. De esta forma, una tarea podría llamar a un punto de entrada de otra tarea aunque ésta no sea local ni esté definida en la parte externa de una unidad de biblioteca.

Sin embargo, para Volz [VOL89] el problema del contexto recursivo (en las consideraciones expuestas este problema se ha englobado dentro del problema más general de implementar el *cactus stack*) junto con la terminación de las tareas, que se analizará en los siguientes apartados, son los dos aspectos más problemáticos de implementar eficientemente en un entorno donde los procesadores no comparten memoria. Las soluciones propuestas por Volz evitan estos problemas restringiendo los datos que pueden compartir las tareas a las variables definidas en la parte más externa de una unidad de biblioteca (variables de carácter estático) ya que su dirección es conocida en tiempo de compilación. Así, una tarea sólo puede referenciar objetos estáticos u objetos locales. Hay que recordar que las tareas son también objetos y, por lo tanto, con esta restricción una tarea sólo puede llamar a una tarea hija (referencia local) o a una tarea definida en la parte más externa de una unidad de biblioteca.

Esta solución no considera problemática la utilización de variables compartidas para comunicarse entre tareas siempre que sean de carácter estático.

La propuesta de Volz simplifica la gestión de la memoria frente al modelo original de Ada, pero no sólo en máquinas sin memoria compartida sino en cualquier tipo de entornos, incluidas máquinas con un único procesador. Sin embargo, esta restricción modifica considerablemente el modelo estructurado en bloques de las tareas Ada, dando como resultado un modelo similar al de Concurrent C.

Puede ser interesante realizar una comparación entre los modelos seguidos por Ada y Concurrent C en cuanto a la compartición de datos entre tareas, en el caso de Ada, y procesos, en el caso de Concurrent C.

Los modelos de gestión de memoria de Concurrent C y Ada se inspiran en los de C y de Algol (estructurados en bloques) respectivamente, extendiéndolos a las construcciones concurrentes de cada lenguaje.

La gestión de la memoria en los lenguajes secuenciales estructurados en bloques es más compleja que en los lenguajes secuenciales basados en la filosofía de C, ya que en estos últimos no es necesario conocer la traza de ejecución para resolver las referencias a las variables. Sin embargo, el modelo *block-structured* facilita el desarrollo de programas más modulares y estructurados.

Estas consideraciones se pueden extender a los lenguajes concurrentes. Así, la gestión de memoria en Ada basada en el *cactus stack* es compleja de implementar en cualquier tipo de entorno [ARD87] [GEH89]. En cambio, la gestión de memoria en Concurrent C es bastante más sencilla ya que cada proceso puede referenciar únicamente objetos locales u objetos cuya dirección se conoce en tiempo de compilación. Como en el caso de los lenguajes secuenciales, existe una contrapartida entre la potencia y flexibilidad del lenguaje, y la facilidad de implementarlo.

La propuesta de Volz para restringir Ada crea un modelo de memoria muy similar al de Concurrent C, pero existe una importante diferencia entre ambos modelos. En el modelo restringido de Ada las tareas pueden comunicarse utilizando variables compartidas. En Concurrent C, sin embargo, los procesos no pueden comunicarse mediante variables compartidas si se desea que el programa sea portable. Así, una determinada implementación puede permitir que los procesos utilicen variables compartidas para comunicarse, sin embargo, otra implementación, por ejemplo sobre un entorno sin memoria compartida, puede no permitirlo.

En resumen, se pueden plantear cuatro propuestas en la utilización de Ada en máquinas sin memoria compartida:

- Implementar el modelo de memoria de Ada. Esto implica permitir la comunicación mediante variables compartidas e implementar el *cactus stack*.

- Implementar un modelo restringido en el que las tareas no pueden comunicarse mediante variables compartidas, pero pueden llamar a puntos de entrada de tareas aunque no sean locales ni estén definidas en la parte externa de una unidad de biblioteca. Con esta propuesta es necesario implementar el *cactus stack* para resolver la llamadas a tareas cuya dirección no se conoce en tiempo de compilación. Esta alternativa es interesante ya que, en nuestra opinión, se eliminan los problemas asociados a la comunicación mediante variables compartidas manteniendo un modelo de tareas que sigue la filosofía de Ada.

- Implementar un modelo restringido en el que las tareas pueden referenciar tareas y variables compartidas siempre que su dirección pueda determinarse en tiempo de compilación o sean locales. Con esta alternativa, que recoge la propuesta de Volz, no hay que implementar el *cactus stack*. Sin embargo, es necesario proporcionar el acceso remoto desde una tarea a una variable. Con esta propuesta se mantiene el uso de variables compartidas para comunicarse pero no se respeta la filosofía del modelo de tareas de Ada.

- Implementar un modelo restringido en el que los únicos objetos no locales que pueden referenciar las tareas son tareas definidas en la parte externa de una unidad de biblioteca. Esta alternativa elimina la comunicación mediante variables compartidas y la necesidad del *cactus stack* creando un modelo de tareas similar al de Concurrent C.

4.4 Terminación de las tareas Ada

La gestión de la terminación de las tareas Ada implica que se produzcan una serie de interacciones que no son visibles por el programador. En este apartado se analizan las características de estas interacciones y sus repercusiones en la eficiencia.

Ada presenta un modelo de terminación de tareas flexible y potente. Sin embargo, algunas características del mismo pueden ser difíciles de implementar eficien-

temente. Antes de realizar un análisis de esta problemática se exponen brevemente las reglas que gobiernan la terminación de las tareas.

- El *master* de una tarea creada a partir de un asignador es la tarea, el bloque, el subprograma o el paquete de biblioteca que contiene la declaración del tipo acceso correspondiente a dicha tarea.

- En cualquier otro caso, el *master* de una tarea es la tarea, el bloque, el subprograma o el paquete de biblioteca que contiene la declaración del objeto que crea la tarea.

Una tarea, además del *master* "directo", puede tener varios masters "indirectos" formando una jerarquía de *masters*. Así, si el *master* de una tarea es, a su vez, una tarea que depende de otro *master*, la tarea también depende de este *master*. La aplicación recursiva de esta regla da lugar a la jerarquía de *masters*.

La reglas para la terminación de una tarea son las siguientes:

- Si una tarea no tiene tareas dependientes termina cuando se ha completado su ejecución.

- Si una tarea tiene tareas dependientes termina cuando se ha completado y todas sus dependientes han terminado.

- Si una tarea está esperando en una alternativa **terminate** de una sentencia **select**, dicha tarea termina cuando depende de algún *master* (directo o indirecto) cuya ejecución está completada y, además, todas las tareas dependientes de este *master* están terminadas o esperando en alternativas **terminate**. Además de dicha tarea terminan todas las tareas dependientes del *master*. A esta situación se le suele denominar onda de terminación.

La ejecución de un bloque o un subprograma que tengan tareas dependientes, por otro lado, no puede ser abandonada hasta que todas las tareas dependientes hayan terminado o estén esperando en una alternativa **terminate**.

La inclusión de la alternativa **terminate** facilita considerablemente la programación de las tareas servidoras, sin embargo, presenta problemas a la hora de ser implementada eficientemente. La inclusión de esta alternativa presenta, al menos, dos dificultades:

- La detección de que una tarea puede terminar no depende únicamente del estado de las tareas hijas (dependientes directas) como ocurriría si no existiese la alternativa **terminate** en el lenguaje, sino que depende del estado de todas sus dependientes, así como del estado de sus *masters* y de las tareas dependientes de dichos *masters*. Esto va a complicar considerablemente la gestión de la terminación de una tarea.

- Mientras que la transición de una tarea al estado de completada es irreversible, el paso al estado correspondiente a esperar en una alternativa **terminate** es, evidentemente, reversible. Esto puede dar lugar a que se produzcan condiciones de carrera en la detección de que una tarea ha terminado. Así, puede ocurrir que se decida que una tarea T1 debe terminar basándose en que otra tarea T2 esté esperando en una alternativa **terminate** pero, desde el momento en que se comprueba el estado de T2 hasta que se decide que T1 debe terminar, el estado de T2 puede haber cambiado debido a que otra tarea T3 haya hecho una llamada a un punto de entrada T2 incluido en el **select** con la alternativa **terminate**.

La mayoría de las implementaciones de la terminación de tareas en un entorno monoprocesador (un único procesador y por lo tanto modelo corrutina) se basan en un supervisor [BAK85] que "congela" la ejecución de las tareas del programa mientras evalúa la posibilidad de que algunas tareas hayan terminado. Las operaciones del supervisor, por lo tanto, no pueden ser interrumpidas por ninguna tarea. De esta forma se asegura que no se producirán condiciones de carrera en la detección de la terminación de las tareas. La solución propuesta en [BAK85] se basa en asociar a cada *master* un contador que indica el número de tareas dependientes directas que no están ni terminadas ni esperando en una alternativa **terminate**. Cuando el contador

llega a cero y el *master* está completado terminarán todas las tareas dependientes de dicho *master*. Si el contador vale cero y el *master* está esperando en una alternativa **terminate** (el *master* debe ser una tarea) el supervisor debe propagar esta actualización hacia arriba en la jerarquía de *masters* hasta que encuentre un *master* cuyo contador sea distinto a cero o cuyo contador sea cero y esté completado. En este último caso todas las dependientes de dicho *master* deben ser terminadas.

Terminación en máquinas con memoria compartida

La aplicación directa de soluciones basadas en un supervisor central a máquinas con múltiples procesadores (modelo multirrutina) y con memoria compartida, puede crear importantes cuellos de botella en el aprovechamiento del paralelismo del sistema ya que estas soluciones se basan en "congelar" la ejecución del programa mientras se analiza el estado de las tareas.

En [FLY87] se plantea un algoritmo basado en el propuesto en [BAK85] eliminando la necesidad de un supervisor central y de las secciones críticas que proporciona el mismo. El algoritmo distribuye el trabajo de detección entre las tareas implicadas. La última tarea que decrementa el contador y obtiene el valor cero inicia la terminación de las tareas correspondientes. El único requisito de sincronización necesario para el algoritmo es que la máquina proporcione una operación atómica del tipo *leer-y-actualizar* para evitar las condiciones de carrera en la gestión de los contadores asociados a cada *master*. No es necesario ningún otro mecanismo de sincronización entre las tareas ya que el algoritmo está diseñado para evitar las condiciones de carrera que pueden presentarse entre el momento en que se consulta el estado de una tarea y el momento en que se decide la terminación de un grupo de tareas. El algoritmo intenta también minimizar la propagación de actualizaciones a través de la jerarquía de *masters* para que la mayor parte de las actualizaciones afecten únicamente al *master* directo de la tarea. Este objetivo se ha conseguido a partir del estudio de algunas propiedades que presenta el modelo de terminación de Ada (p.ej. si un *master* **m** está esperando en una alternativa **terminate** y todos sus dependientes están

terminados o esperando en una alternativa **terminate,** no puede existir ninguna llamada a un punto de entrada de una tarea dependiente de **m** ni pueden crearse nuevas tareas dependientes de **m**).

El algoritmo resuelve también los problemas específicos que presenta la terminación de bloques: La ejecución de un bloque puede ser abandonada sin haber completado su ejecución. Esta situación puede ocurrir cuando un bloque, incluido en el cuerpo de una tarea, contiene un **select** con una alternativa **terminate** correspondiente a dicha tarea. Si la ejecución del bloque se queda parada en dicho **select** y existe un *master* de la tarea en la que está incluido el bloque que cumple las condiciones de terminación, la ejecución del bloque será abandonada aunque éste no se haya completado. Estos problemas no se presentan con los subprogramas ya que no pueden contener sentencias **select** y, por lo tanto, sólo se puede abandonar su ejecución cuando estén completados y sus tareas dependientes estén terminadas o esperando en una alternativa **terminate.**

Terminación en máquinas sin memoria compartida

En máquinas sin memoria compartida no puede existir una estructura de datos accesible por todos los procesadores que refleje el estado de las tareas del programa. En su lugar, los procesadores deben intercambiar mensajes entre sí para conocer el estado de las tareas en ese instante. La falta de esta estructura global aumenta la posibilidad de que se produzcan condiciones de carrera en la detección de la terminación [VOL89].

La solución propuesta en [FLY87] para máquinas con memoria compartida puede ser adaptada, según sus autores, a máquinas sin memoria compartida. En [BUR87] se recoge un algoritmo, propuesto por Raymond, específico para máquinas sin memoria compartida. El algoritmo está basado en la propagación de dos ondas de mensajes: Una primera onda de peticiones de terminación dirigida hacia abajo en la jerarquía de *masters* y una segunda onda de respuestas dirigidas hacia arriba. Cuando un *master* se completa genera una onda de peticiones de terminación que se propaga

a todas sus tareas dependientes. El *master* debe esperar un mensaje de "lista para terminar" de cada una de sus dependientes. Una tarea dependiente responderá con un mensaje de "lista para terminar" cuando esté terminada (si la tarea tiene dependientes para llegar al estado de terminada ha debido propagar, anteriormente, su propia onda de terminación y haber recibido los mensajes de "listas para terminar" correspondientes) o esté esperando en una alternativa **terminate** y todas sus dependientes estén terminadas o esperando en una alternativa **terminate**. Puede ocurrir que una tarea esperando en una alternativa **terminate** mande un "lista para terminar" al *master* correspondiente y, posteriormente, reciba una llamada a un punto de entrada abierto con lo que deja de estar lista para terminar. En este caso la tarea que llamó al punto de entrada debe mandar un mensaje de "no lista para terminar" al *master* para evitar la posible condición de la carrera en la detección de la terminación.

Independientemente del algoritmo utilizado se pueden distinguir dos clases de mensajes:

- Mensajes que reflejan el cambio de estado de la tarea y que permiten llevar a cabo al algoritmo correspondiente. Dicho algoritmo debe diseñarse para minimizar el número de mensajes necesarios.

- Mensajes que permiten la propagación de la onda de terminación a un conjunto de tareas cuando éstas hayan alcanzado las condiciones de terminación. Esta onda puede afectar a un gran número de tareas.

Por lo tanto, la implementación del modelo de terminación de Ada en máquinas sin memoria compartida puede implicar un considerable número de mensajes, sobre todo si el "árbol" de tareas del programa es complejo, lo que dificulta la ejecución eficiente del programa en este tipo de entornos. Estas dificultades son debidas a la inclusión de la alternativa **terminate** en el lenguaje. Una posibilidad, por lo tanto, sería eliminar esta alternativa del lenguaje, pero esta solución restringiría considerablemente la potencia del modelo de terminación de Ada.

Modelo simplificado de terminación

Una alternativa interesante es la adoptada en el lenguaje Concurrent C. En éste se incluye la alternativa **terminate** pero su semántica es muy diferente a la de Ada [GEH88]: Cuando todos los procesos de un programa están completados o esperando en una alternativa **terminate**, el programa entero termina. Inicialmente Concurrent C incluía un mecanismo de terminación similar al de Ada pero, posteriormente, optó por la solución simplificada por las siguientes razones [CME89]:

- El modelo de terminación inicial (estilo Ada) es muy difícil de implementar eficientemente en máquinas sin memoria compartida.
- El modelo inicial es difícil de entender por los usuarios.
- La mayoría de los programas de usuario estudiados por los autores no necesitan el modelo inicial, sino que es suficiente la solución simplificada.

El mecanismo de terminación de Concurrent C es menos potente que el de Ada y, por lo tanto, existen aplicaciones en las que la solución usando Ada será más sencilla de programar que la de Concurrent C. Sin embargo, los diseñadores del lenguaje han sacrificado esta pérdida de potencia por una mayor facilidad en la implementación.

En [GEH89] se expone brevemente como se podría implementar este modelo de terminación simplificado. La propuesta está basada en que el Entorno de Ejecución cree un proceso de baja prioridad que compruebe periódicamente el estado de los procesos del programa. Cuando se detecte que todos los procesos están completados o esperando en una alternativa **terminate** el programa terminará.

Este tipo de solución sólo serviría para entornos uniprocesador. En entornos con múltiples procesadores, ya sea con o sin memoria compartida, este tipo de solución no será válido debido a que, aunque la alternativa **terminate** de Concurrent C

simplifica la gestión de la terminación frente a la de Ada, no evita las condiciones de carrera potenciales en la detección de terminación.

La solución en este tipo de entornos debe estar basada en detectar cuando el programa alcanza un estado de "estabilidad", esto es, cuando todos los procesos del programa están completados o esperando una alternativa **terminate**. Este estado de "estabilidad" puede detectarse manteniendo un contador que refleje, en cada momento, el número de procesos que ni están completados ni esperando una alternativa **terminate**. Cuando este contador llega a cero el programa completo terminaría.

Es interesante observar que en vez de un contador por cada *master*, como ocurría en el caso de Ada, sólo es necesario un único contador lo cual facilita considerablemente la implementación.

En sistemas con memoria compartida será necesario asegurar el acceso exclusivo a dicho contador para evitar las condiciones de carrera. Este acceso exclusivo puede ser proporcionado a nivel hardware, si éste provee operaciones atómicas del tipo leer-y-actualizar.

En sistemas sin memoria compartida la gestión del contador se llevaría a cabo mediante el intercambio de mensajes entre los distintos procesadores que ejecutan los procesos Concurrent C. Pueden surgir, como se expuso anteriormente en el caso Ada, condiciones de carrera en la detección de la terminación. Si un proceso P1 esperando en una alternativa **terminate** es llamado por un proceso P2 y después P2 se completa, puede ocurrir que el mensaje informando de que P2 se ha completado llegue antes que el mensaje que informa que P1 ha dejado de esperar en la alternativa **terminate** y, por lo tanto, se pueda tomar la decisión de terminar el programa erróneamente. Para evitar esta anomalía se podría plantear una solución similar a la propuesta por Raymond para Ada. Así, P2 después de realizar la cita mandaría un mensaje informando de que P1 ya no está esperando en una alternativa **terminate**.

Una de las ventajas del modelo simplificado es que desaparece la onda de propagación de terminación y, por consiguiente, los mensajes que conlleva la misma.

Con este modelo una tarea esperando una alternativa **terminate** sólo puede pasar a terminada cuando termina todo el programa.

Puede ser interesante, dentro del proceso de revisión de Ada 9X, evaluar la posibilidad de adoptar un modelo de terminación similar al de Concurrent C para Ada. Esta evaluación debería llevarse a cabo teniendo en cuenta dos factores principalmente:

- La pérdida "real" de potencia que implica el modelo simplificado. En otras palabras, estudiar si existen aplicaciones en las que la utilización del modelo simplificado penaliza considerablemente su diseño.

- La mejora en eficiencia que implica el modelo simplificado en entornos sin memoria compartida. Como se expuso anteriormente la implementación se simplifica, pero, aun así, dista mucho de ser trivial.

Terminación de tareas dependientes de paquetes de biblioteca

Por último hay que resaltar que el MRL no define cuando deben terminar las tareas que dependen de un paquete de biblioteca. Así, una implementación puede terminar la ejecución de un programa cuando el procedimiento principal esté listo para terminar (esté completado y sus tareas dependientes estén terminadas o esperando en una alternativa **terminate**) aunque las tareas dependientes de paquetes de biblioteca no estén completadas. Otra alternativa, más razonable, es esperar hasta que las tareas dependientes de paquetes de biblioteca terminen su trabajo [BUR85].

Esta indefinición en la terminación de tareas dependientes de paquetes de biblioteca se debe a que no encajan en el modelo general de terminación de Ada. Esta tareas dependen de un *master* que no es una unidad activa y, por lo tanto, no tiene sentido hablar de que el *master* está completado.

4.5 La implementación de las llamadas condicionales y temporizadas

La implementación, en un entorno en el que no existe memoria compartida entre los procesadores, de algunas de las primitivas de Ada relacionadas con el tiempo plantea cuestiones interesantes. Volz y Mudge [VOL87] estudian exhaustivamente estas cuestiones y proponen algunas soluciones a los problemas encontrados.

En dicho artículo se estudian solamente las llamadas condicionales a puntos de entrada y las llamadas temporizadas a puntos de entrada. No se ocupa del resto de las primitivas relacionadas con el tiempo (**select temporizados** y **condicionales**, y **delays**) ya que sus acciones son locales y afectan únicamente al procesador que esté ejecutando la tarea correspondiente. Las llamadas condicionales y temporizadas, en cambio, basan su acción en el estado de la tarea a la que llaman. Si existe memoria compartida entre los procesadores que ejecutan ambas tareas, la tarea que llama averigua el estado de la tarea llamada consultando alguna estructura de datos que resida en la memoria compartida. Si no existe memoria compartida entre los procesadores deben generarse una serie de mensajes entre ambos procesadores para determinar el estado de la tarea llamada. Existen dos diferencias principales entre ambos casos:

- Sin memoria compartida el procesador asociado con la tarea llamada está implicado en la ejecución de la llamada, mientras que si existiera memoria compartida no estaría implicado.

- El tiempo que tarda en realizarse la consulta si no existe memoria compartida será considerablemente más grande.

Existen problemas de interpretación del MRL cuando se refiere a llamadas condicionales y temporizadas:

- *Llamadas condicionales.* El MRL afirma que una llamada condicional a un punto de entrada será cancelada si la cita no puede ser realizada inmediatamente. Si se interpreta el término "inmediato" en el sentido temporal, de manera estricta, ninguna llamada tendrá, lugar ya que en averiguar el estado de la tarea llamada se tarda algún tiempo, más aún si no existe memoria compartida.

Con esta interpretación las llamadas condicionales pierden su sentido. Sin embargo, existen otros párrafos en el MRL que hacen una interpretación de la llamada condicional que no depende del tiempo sino del estado de la tarea llamada. Con esta última interpretación, teniendo en cuenta que una llamada a un punto de entrada con una temporización menor o igual que cero es equivalente a una llamada condicional, se presenta, en entornos sin memoria compartida principalmente, una curiosa singularidad. Si **d** es el tiempo total que tarda en realizarse el intercambio de mensajes entre los procesadores implicados, y se supone que la tarea llamada está lista para aceptar una cita, se da la siguiente situación:

- Las llamadas temporizadas con plazos menores o iguales que cero tendrán éxito, ya que son equivalentes a llamadas condicionales que con la última interpretación son independientes del tiempo.

- Las llamadas temporizadas con plazos mayores que cero y menores que **d** fallarán.

- Las llamadas temporizadas con plazos mayores que **d** tendrán éxito.

En el caso de memoria compartida también se tarda cierto tiempo en averiguar si la tarea llamada está lista para aceptar la cita pero, en general, este tiempo va a ser considerablemente más pequeño e incluso puede ser inferior a la precisión del **delay** en algunas implementaciones.

- *Llamadas temporizadas.* El MRL contiene las dos siguientes afirmaciones sobre las llamadas temporizadas:

  - Si se puede iniciar una cita dentro del intervalo especificado, se realiza ésta.

  - Si el plazo ha expirado se cancela la llamada al punto de entrada.

En estos estamentos está implícita la necesidad de un sentido único del tiempo en el sistema, al menos dentro de unos márgenes aceptables. Si en el sistema

existen múltiples relojes, conseguir un sentido único del tiempo es un problema complejo. En [DEN90] se plantean algunos algoritmos para resolverlo.
Los dos estamentos del MRL pueden llevar a interpretaciones contradictorias cuando se implementan llamadas temporizadas en entornos sin memoria compartida. Si PV_1 es el procesador que ejecuta la tarea que llama y PV_2 es el procesador que ejecuta la tarea llamada se pueden distinguir dos opciones:

- PV_2 espera hasta el final del intervalo que la tarea llamada esté lista para aceptar la cita. Si terminado el intervalo la tarea no está preparada manda un mensaje a PV_1 informándole de que la cita no puede tener lugar. Pero esta situación viola el segundo estamento ya que PV_1, debido al retardo en la transmisión del mensaje, no podrá cancelar la llamada hasta un cierto tiempo después de la conclusión del intervalo.

- PV_1 recibe, en la conclusión del intervalo, el mensaje de PV_2 informándole que la cita no puede tener lugar. Esto implica que PV_2 ha mandado el mensaje cierto tiempo antes y, por lo tanto, no ha esperado hasta el final del intervalo que la tarea llamada pase a estar lista para aceptar la cita. Esta situación viola el primer estamento.

Como no pueden cumplirse simultáneamente ambos estamentos, se puede hacer dos interpretaciones diferentes de la llamadas temporizadas dependiendo de cual de ellos se ha respetado. Se presentan cuatro casos dependiendo de la interpretación seleccionada y de qué procesador (PV_1 o PV_2) decida si la cita será realizada:

- Existen dos casos, decisión en PV_1 manteniendo el estamento 1 y decisión en PV_2 manteniendo el estamento 2, cuya implementación depende de que el tiempo de transmisión de los mensajes esté acotado. Esta cota, en muchos casos, es muy difícil de conseguir por lo que estas soluciones no suelen ser utilizadas.

- Decisión en PV_2 manteniendo el estamento 1. La temporización se realiza en PV_2. Se necesitan solamente dos mensajes: PV_1 manda un mensaje para iniciar la interacción y PV_2 le devuelve un mensaje informando si la cita ha podido ser iniciada. Esta es la solución adoptada en el artículo.

- Decisión en PV_1 manteniendo el estamento 2. La temporización se realiza en PV_1. Se necesita otro mensaje ya que una vez que PV_1 recibe la notificación de PV_2 de que la tarea llamada está dispuesta a realizar la cita, debe mandar, dependiendo de si la notificación ha llegado antes del fin del intervalo o no, un mensaje a PV_2 informándole de la decisión tomada. Esta solución es adoptada en [ARE88].

El problema revisado en este apartado es más un problema semántico que un problema de eficiencia. Es necesario que el MRL defina con más precisión la semántica de estas llamadas en este tipo de entornos. De todas formas, se ha incluido este análisis dentro del estudio de la problemática de implementar Ada por considerarla de interés.

4.6 Sumario del capítulo

Para que un determinado lenguaje sea adecuado para la programación de máquinas paralelas debe poder implementarse eficientemente en diferentes tipos de máquina. El objetivo del capítulo era analizar si el lenguaje Ada cumplía este requisito estudiando las características del lenguaje que puedan ser más problemáticas de implementar con eficiencia.

El análisis se ha basado en evaluar las necesidades de comunicación que conlleva la máquina virtual Ada, esto es, su granularidad, no sólo debidas a las interacciones entre las tareas especificadas por el programador, sino también a las interacciones implícitas introducidas por la implementación de algunos mecanismos del lenguaje. Estas necesidades de comunicación pueden sobrepasar a las proporcionadas por la máquina sobre la que se implementa y, por lo tanto, pueden impedir su utilización eficiente.

La mayor parte del estudio se ha centrado en la implementación de Ada sobre máquinas con memoria distribuida ya que éstas presentan mayor granularidad siendo, por consiguiente, más problemáticas.

El primer aspecto estudiado ha sido la comunicación entre tareas usando variables compartidas. La máquina Ada presenta un modelo de memoria compartida. Para poder implementar de manera eficiente una máquina con un modelo de memoria compartida sobre una máquina con memoria distribuida es necesario que los accesos a la memoria compartida sean de ciclo partido, de forma que una vez iniciado el acceso se produzca un cambio de contexto y se aproveche al máximo el procesador.

El entorno de ejecución Ada debe interceptar de alguna forma los accesos a la memoria compartida y, a continuación, realizar un cambio de contexto y mandar el mensaje al procesador correspondiente.

Sin embargo, Ada no presenta un modelo de memoria compartida "puro" ya que permite que se creen copias locales de las variables compartidas, las cuales sólo deben ser actualizadas en ciertos puntos de sincronización. Esta facilidad disminuye las necesidades de comunicación de la máquina Ada y puede posibilitar su implementación en máquinas con memoria distribuida sin necesidad de utilizar transferencias de ciclo partido.

Se ha realizado un análisis intensivo del mecanismo de copias locales identificando la problemática asociada a su gestión. En primer lugar, no es posible conocer en tiempo de compilación qué variables compartidas hay que actualizar cuando se llega a un punto de sincronización. Tanto las sentencias que impliquen dos o más caminos alternativos como los subprogramas impiden que esta gestión se realice en tiempo de compilación, a no ser que se introduzcan nuevos puntos de sincronización además de los definidos por el lenguaje, lo cual puede aumentar considerablemente las comunicaciones requeridas.

Frente a las soluciones estáticas surgen alternativas que realizan la gestión en tiempo de ejecución basándose en información que se almacena con cada copia local.

Se plantean varios algoritmos que almacenan diferentes tipos de información y que buscan minimizar las sincronizaciones. Sin embargo, las soluciones dinámicas siguen necesitando que las llamadas y retornos de subprogramas sean puntos de sincronización. Una interesante solución que elimina estos problemas y acelera la gestión de las copias locales es la utilización de memorias cache.

Las memorias cache con esquemas de coherencia software son muy adecuadas para la gestión de las copias locales de Ada. Se exponen algunos algoritmos para mantener la coherencia de las caches. Hay que resaltar que aunque la utilización de caches elimina la necesidad de que las llamadas y retornos de subprogramas sean puntos de sincronización, en una máquina con memoria distribuida el subprograma y la tarea que lo invoca pueden ser ejecutados por dos procesadores distintos por lo cual se deberían mantener la invocación y el retorno como puntos de sincronización.

En segundo lugar, se han analizado las repercusiones que trae consigo, en cuanto a interacciones, el mantenimiento de las reglas de ámbito de Ada. Esta necesidad implica que la unidad que activa una tarea o un subprograma debe suministrarle información de contexto que le permita resolver las referencias a variables que no son locales y cuya dirección no se conoce en tiempo de compilación. Si ambas unidades son ejecutadas por dos procesadores distintos será preciso que se realice una interacción entre ambos.

La frecuencia de estas interacciones, así como sus características, nos inclinan a considerar que este aspecto no afecta de forma importante a la implementación eficiente de Ada. De todas formas, se han revisado las diversas opiniones diferentes sobre este tema.

El tercer aspecto analizado es la terminación de las tareas. El modelo de terminación de Ada es muy potente pero genera una gran cantidad de interacciones, debido principalmente a la existencia de la alternativa **terminate**. Ante esta situación, se ha planteado un modelo simplificado siguiendo la filosofía del Concurrent C.

El último aspecto analizado es la implementación de llamadas condicionales y temporizadas en máquinas con memoria distribuida. Aunque no se trate de un problema de eficiencia sino de semántica, nos ha parecido interesante incluir una exposición de esta problemática.

5

Implementación de Ada en diferentes entornos

5.1 Introducción

En este capítulo se realiza un análisis de la implementación de Ada sobre los tres tipos de máquinas existentes con respecto al modelo de interacción. Este análisis, que se apoya en las conclusiones obtenidas en el capítulo anterior, permite identificar las dificultades de implementar Ada en entornos sin memoria compartida.

Ante estas dificultades se han realizado diversas propuestas para modificar el lenguaje. Se revisarán estas propuestas centrándose principalmente en la realizada dentro del proyecto de revisión del lenguaje Ada 9X. Se examinará la manera en que estas propuestas solventan los problemas asociados a la implementación del lenguaje en máquinas sin memoria compartida.

El lenguaje Ada fue diseñado para ejecutarse tanto en máquinas desnudas como en máquinas con sistema operativo. Se estudiará en este capítulo la problemática específica de implementar el lenguaje sobre el sistema operativo.

Todas las consideraciones realizadas en el capítulo se basarán en la utilización de Ada siguiendo una estrategia de tipo *programa único*. Existe otra alternativa de-

nominada *programas múltiples* en la que para programar una máquina paralela se utilizan varios programas que se comunican entre sí en vez de un único programa [BUR87] [KEE85]. No se estudiará, en principio, este tipo de estrategia.

A pesar de su importancia no se analizarán los aspectos relacionados con la tolerancia a fallos en Ada. En [KNI87] se desarrolla un interesante análisis sobre esta problemática.

5.2 Partición y configuración

Cuando se desea ejecutar un programa en una máquina paralela es necesario realizar dos operaciones: Partición y configuración.

La partición consiste en dividir el programa en un conjunto de unidades que pueden ejecutarse en paralelo. La configuración es la asignación de estas unidades a los procesadores existentes.

En Ada, al igual que en todos los lenguajes descendientes del CSP, la partición está explícita en el programa [SAR89], las tareas son las unidades que pueden ejecutarse en paralelo. Se podría, evidentemente, seleccionar otros elementos como unidades de partición, sin embargo este trabajo se centra sobre el paralelismo explícito, esto es, el que introduce el programador de forma explícita y, en Ada, el programador introduce paralelismo mediante la utilización de tareas. Por su parte, el MRL identifica a la tarea como la unidad de partición, aunque deja abierta la posibilidad de que diferentes partes de una tarea sean ejecutadas en paralelo.

Bajo este supuesto, la partición de un programa Ada viene dada directamente por el conjunto de tareas definidas por el usuario. Estas tareas identifican, siguiendo nuestro modelo jerárquico de paralelismo, los procesadores virtuales de la máquina Ada necesarios para ejecutar el programa.

La operación de la configuración, por otra parte, incluye la asignación de las tareas (procesadores virtuales Ada) a los procesadores virtuales, ya sean procesos del

sistema operativo o procesadores físicos, de la máquina sobre la que se ejecutará el programa.

Existen diferentes alternativas a la hora de realizar esta asignación:

- Asignación explícita dentro del programa. El programador especifica en el código del programa a qué procesador queda asociado un determinado proceso. Este modelo aparece en Occam 2 mediante la sentencia **PLACED PAR**. En Concurrent C el operador **create** permite crear un proceso asignándolo a un determinado procesador. Ada no incluye ningún mecanismo de este tipo aunque existen algunas propuestas en esta línea. En [KRI88] se plantea la utilización del **pragma** SITE para llevar a cabo esta función.

- Asignación explícita fuera del programa. Otra alternativa es separar la información de configuración del resto del programa creando una herramienta que permita al usuario especificar la configuración del programa en una determinada máquina. El APPL (Ada Program Partitioning Language) [JHA89b] cumple en parte esta función. Realmente esta herramienta no sólo permite realizar la configuración sino también la partición. Esto es debido a que los autores del APPL asumen, en principio, que varias construcciones Ada, además de la tarea, pueden ser unidades de partición (*fragmentos* en terminología APPL). Así, el usuario utilizando APPL puede definir distintos fragmentos dentro de un programa y asignarlos a diferentes procesadores. La alternativa soportada por APPL se suele denominar *post-partición* ya que la partición y configuración se realiza una vez escrito el programa.

- Asignación automática. En las dos alternativas anteriores se dejaba la configuración en manos del programador, lo cual puede ser necesario en algunos casos. Por ejemplo, un proceso que controla un sensor debe asignarse al procesador que tiene asociado dicho sensor. Sin embargo, en muchos casos y especialmente cuando existe un número elevado de procesadores, es necesario que la asignación se realice de forma automática. Esta configuración automática debe analizar

las dependencias entre las tareas del programa para buscar una asignación que permita una utilización eficiente de la máquina. Como se expuso en el segundo capítulo esta labor es difícil, sobre todo con un esquema de dependencias entre tareas tan complejo como el de Ada. Quizás por ello los estudios sobre este tema se han centrado en la asignación explícita.

- Asignación automática en tiempo de ejecución. Con esta alternativa la asignación entre procesos y procesadores se realiza dinámicamente en tiempo de ejecución. Este tipo de asignación recibía en nuestra terminología el nombre de asignación dinámica, mientras que las tres anteriores alternativas se clasificaban como asignaciones estáticas. Hay que recordar que la asignación dinámica sólo puede realizarse directamente en máquinas con memoria compartida en las que el código de los procesos sea directamente accesible desde todos los procesadores. En este caso la operación de configuración se simplifica considerablemente, cada procesador puede seleccionar en cada momento entre cualquier proceso que esté en estado ejecutable. En caso contrario la asignación dinámica implica la migración de los procesos, lo que resulta normalmente poco eficiente.

5.3 Implementación de Ada estándar

En este apartado se van a exponer algunas consideraciones sobre la implementación de Ada estándar, sin restricciones ni modificaciones, en máquinas que sigan los tres modelos de interacción definidos, a saber, memoria compartida, memoria distribuida y con modelo mixto. Esta exposición se apoya en las conclusiones alcanzadas en el capítulo anterior.

5.3.1 Implementación sobre máquinas con memoria compartida

Se va a distinguir entre las máquinas que permiten realizar directamente asignación dinámica y las que no.

Máquinas que permiten directamente asignación dinámica

Se trata de máquinas en las que el código de los procesos es directamente (uniformemente) accesible desde todos los procesadores. La máquina de nivel físico subyacente va ser normalmente un multiprocesador simétrico (UMA).

El modelo de paralelismo de Ada se adapta perfectamente a este tipo de máquinas. La asignación de tareas a procesadores se puede realizar de forma dinámica en tiempo de ejecución.

En cuanto a los problemas estudiados en el capítulo anterior se resuelven de forma satisfactoria en este tipo de sistemas.

Las variables compartidas se almacenarán en la memoria compartida pudiendo optimizarse su acceso con la utilización de caches, ya sean convencionales o las analizadas en el capítulo anterior. No es necesario implementar accesos a memoria de ciclo partido ya que la latencia en los accesos es baja.

El *cactus stack* puede implementarse gracias a la memoria compartida con una complejidad similar a la que existe en sistemas uniprocesador.

La gestión de la terminación se puede realizar utilizando, por ejemplo, el algoritmo descrito en el capítulo anterior que proporciona una solución bastante eficiente.

En [BUR85] se plantea, dentro del análisis de implementar Ada sobre multiprocesadores con memoria compartida, el cuello de botella que puede surgir en la inicialización de las tareas de un programa en el caso que una tarea deba dar un valor inicial al resto de las tareas. Se plantea un programa que realiza una inicialización en tiempo exponencial. Realmente este problema es independiente del tipo de máquina en la que se ejecute el programa y se debe a que en Ada no se permite dar valores iniciales a una tarea, como en Concurrent C, lo que obliga a que se necesite una cita para dar el valor inicial.

La utilización de cada procesador al no existir accesos de ciclo partido vendrá dada por la latencia en los accesos a la memoria compartida **s** y la granularidad del programa **c** (**U= c/(c+s)**). Dependiendo del perfil de paralelismo del programa y las relaciones de dependencia entre los procesos del mismo se obtendrá la utilización real de cada procesador. El número máximo de procesadores que se utilizarán para ejecutar el programa viene dado por el máximo número de procesos que pueden estar ejecutándose simultáneamente según el perfil de paralelismo del programa. Sin embargo, con este número máximo de procesadores se puede obtener una mala utilización de la máquina si existen fases con poco paralelismo en el perfil del programa.

Máquinas que no permiten directamente asignación dinámica

En este apartado se analizan máquinas en las que el código de los procesos no está directamente (uniformemente) accesible a todos los procesadores, a pesar de que se trate de una máquina con memoria compartida. La máquina de nivel físico subyacente puede ser un multiprocesador de tipo NUMA, en el cual las instrucciones que ejecuta un procesador están almacenadas en su memoria local. Si un procesador ejecutase instrucciones que no estuviesen almacenadas en su memoria local, el rendimiento del sistema disminuiría considerablemente.

La principal diferencia con la anteriores máquinas es que no se puede realizar directamente una asignación en tiempo de ejecución. Se realizará una asignación estática, ya sea por parte del programador o automáticamente, y posteriormente, en tiempo de ejecución, se puede realizar (o no) una asignación dinámica mediante la migración de procesos. Sin embargo, la migración de procesos suele ser una operación poco eficiente.

De esta forma, con una solución puramente estática puede ocurrir que existan procesadores sin hacer trabajo útil mientras hay procesos en estado ejecutable asociados a otros procesadores. Con la migración de procesos se puede aliviar este problema pero con el coste de realizar esta operación.

Por otra parte, la latencia en estas máquinas es mayor que en el caso anterior. La latencia va creciendo con el número de procesadores. Si el valor de ésta es grande, no será posible ejecutar de forma eficiente programas con granularidad fina. Ante esta situación se puede utilizar una cache, como por ejemplo las analizadas en el capítulo anterior, para disminuir la latencia, o bien realizar accesos de ciclo partido para esconder la latencia. Estos accesos de ciclo partido junto con los cambios de contexto correspondientes pueden ser implementados por el entorno de ejecución de Ada, sin embargo no se consigue normalmente la suficiente eficiencia en este nivel. Sería necesario que los accesos de ciclo partido y los cambios de contexto se gestionarán en el nivel de la arquitectura de la máquina, esto es, que la máquina utilizada proporcionase, en el nivel de arquitectura, un modelo de memoria compartida pero con acceso de ciclo de partido.

5.3.2 Implementación sobre máquinas con memoria distribuida

En este tipo de máquinas es todavía más complicado realizar asignación en tiempo de ejecución ya que la operación de migrar un proceso es generalmente más lenta que en las anteriores.

Como vimos en el capítulo anterior, la utilización de variables compartidas en este tipo de máquinas lleva a soluciones poco eficientes debido a que la latencia en la comunicación suele ser elevada. Ante esta situación existen como antes dos alternativas: Utilización de copias locales y accesos de ciclo partido.

En este caso, sin embargo, es posible aplicar estas alternativas sin apoyo hardware debido a que la latencia suele ser bastante mayor que en las anteriores máquinas. El entorno de ejecución Ada puede encargarse de la gestión de las copias locales o de los accesos de ciclo partido con los correspondientes cambios de contexto.

La implementación del *cactus stack* no resulta problemática como se analizó anteriormente. Será necesario incluir la información de contexto en los mensajes de activación de una tarea o un subprograma remoto.

La gestión de la terminación de tareas, sin embargo, puede ser problemática en este tipo de entornos ya que puede generarse un número considerable de mensajes si el árbol de tareas es complejo.

Si se utiliza multiplexación y accesos de ciclo partido para esconder la latencia, será necesario que exista el suficiente exceso de paralelismo en el programa respecto al de la máquina.

5.3.3 Implementación sobre máquinas con un modelo mixto

La mayoría de las consideraciones realizadas sobre las máquinas con memoria distribuida pueden aplicarse a este tipo de máquinas con las diferencias debidas a la existencia de memoria compartida entre algunos procesadores.

En cuanto a la configuración, se puede realizar un reparto estático de conjuntos de procesadores que comparten memoria y, posteriormente, llevar a cabo una asignación dinámica dentro de estos conjuntos.

El algoritmo de configuración debería tener en cuenta las características de la máquina para realizar una asignación del programa lo más eficiente posible. De esta forma, tareas que se comunican frecuentemente o que utilizan variables compartidas pueden asignarse al mismo conjunto de procesadores que comparten memoria. La configuración debería realizarse de forma que se simplifique todo lo posible la gestión de la terminación de las tareas.

5.4 Implementación de Ada restringido o modificado. Ada 9X

Debido a las dificultades en la implementación de Ada en determinados entornos, principalmente cuando no existe memoria compartida, se ha planteado tanto la utilización restringida del lenguaje como la inclusión de nuevas construcciones que disminuyan estas dificultades.

La mayoría de las propuestas para implementar Ada en sistemas con memoria distribuida están basadas de alguna manera en la utilización del paquete de biblioteca

como unidad de partición. Un paquete de biblioteca es una construcción pasiva que una vez elaborado e inicializado, sólo se ejecuta cuando los subprogramas definidos en el mismo son invocados por otras unidades. Sin embargo, si existen tareas declaradas dentro de un paquete de biblioteca, éste podría considerarse de alguna forma una entidad activa. Repartiendo paquetes de biblioteca, en vez de tareas, entre los procesadores se eliminan algunas de las características problemáticas de Ada.

La gestión de la terminación de tareas no involucra a tareas pertenecientes a distintas particiones y, por lo tanto, se puede realizar de forma independiente en cada partición.

Por otro lado, desde una partición sólo se pueden referenciar objetos estáticos declarados en otras particiones, eliminando la necesidad de que el *cactus stack* atraviese particiones. De esta forma, cuando se activa una unidad remota no es necesario mandarle información de contexto.

En [VOL89] y [MUD87] se recoge esta propuesta. No se restringe, sin embargo, la utilización de variables compartidas que era en nuestro análisis uno de los factores que podían causar ineficiencias al implementar Ada en sistemas sin memoria compartida.

Con APPL [JHA89a] [JHA89b] se podía descomponer un programa ya escrito en una serie de fragmentos y luego asignarlos a los procesadores existentes. Aunque en principio el programador puede seleccionar fragmentos muy diversos, en la implementación descrita en el artículo sólo pueden seleccionarse como fragmentos (unidad de partición) unidades de biblioteca y entidades declaradas en la parte visible de paquetes de biblioteca. Con esta restricción la terminación de tareas se realiza de forma independiente en cada fragmento ya que las tareas dependientes de una tarea dada estarán en el mismo fragmento.

Tampoco es necesario que el *cactus stack* cruce a través de los fragmentos ya que desde un fragmento sólo se referencian objetos estáticos cuya dirección se conoce en tiempo de compilación. Como en el caso anterior se eliminan ambos problemas

quedando únicamente el acceso remoto a variables compartidas como posible factor de ineficiencia.

5.4.1 Nodo Virtual

La mayoría de las propuestas se basan en la introducción del concepto de **nodo virtual** como unidad de partición [TED84].

El concepto de nodo virtual aparece en otros lenguajes como SR. Algunas de las características de los nodos virtuales son las siguientes:

- Son unidades de partición del programa.
- Proporciona un interfaz bien definido a otros nodos virtuales del programa.
- Cada nodo virtual consta de uno o más procesos. Los procesos de un mismo nodo virtual pueden comunicarse utilizando memoria compartida. La comunicación de un proceso con otro que corresponda a un nodo virtual diferente se realiza a través del interfaz del nodo utilizando normalmente algún mecanismo de paso de mensajes.

Un lenguaje que incluya una construcción análoga al nodo virtual definirá, como se expuso en el capítulo tres, una máquina virtual con un modelo mixto. En dicha máquina existirán grupos de procesos, de forma que dentro de cada grupo los procesos se comunican mediante memoria compartida, mientras que procesos en distintos grupos no comparten memoria, comunicándose mediante un mecanismo de paso de mensajes.

La introducción del concepto de nodo virtual en Ada modifica el modelo de interacción de Ada. El modelo de memoria compartida original queda sustituido por un modelo mixto.

En la mayoría de las propuestas, un nodo virtual en Ada está formado por un conjunto de unidades de biblioteca que definen una unidad de partición. De alguna manera, la utilización del nodo virtual se puede entender como una generalización

del uso del paquete de biblioteca como unidad de partición que se analizó anteriormente, ya que un nodo virtual está compuesto por un conjunto de unidades (paquetes y subprogramas) de biblioteca. Como consecuencia de ello alguno de los aspectos positivos del uso de paquetes de biblioteca, como son mantener la gestión del *cactus stack* y de la terminación de tareas locales a cada partición, persisten con los nodos virtuales.

En una primera fase se han realizado algunos proyectos para introducir el concepto de nodo virtual en Ada pero sin modificar el lenguaje. Los dos principales desarrollos sobre este tema han sido el proyecto YDA [HUT88] y el proyecto DIADEM [ATK88]. Posteriormente se han realizado propuestas para incluir este concepto como una nueva construcción del lenguaje.

5.4.2 Nodos virtuales en YDA y DIADEM

Ambas soluciones se basan en realizar transformaciones sobre el programa fuente Ada para permitir la ejecución distribuida del mismo usando el nodo virtual como unidad de partición. Existe por lo tanto independencia del compilador. Estas transformaciones realizan, entre otras acciones, la inclusión de las unidades subordinadas (*stubs*) que permiten llevar a cabo las invocaciones remotas.

A este tipo de estrategia, frente a la utilizada en APPL, se la denomina pre-partición ya que desde el momento del diseño del programa se tiene en cuenta cual es la unidad de partición, en ese caso el nodo virtual.

Existen en ambos casos herramientas que permiten al programador especificar las unidades de biblioteca (UB) que forman parte de un determinado nodo virtual y que, además, realizan la comprobación de que dichas unidades cumplen las restricciones correspondientes.

Cada nodo virtual tiene una unidad de biblioteca **raíz**, de forma que el cierre transitivo de las dependencias de la raíz identifica todas las unidades pertenecientes a dicho nodo virtual. Algunas de esta unidades formarán el interfaz del nodo virtual.

Dos nodos virtuales únicamente pueden compartir una unidad de biblioteca cuando esta unidad es el interfaz de uno de los nodos o si se trata de una unidad que puede replicarse en ambos nodos sin cambiar la semántica del programa. A este último tipo de unidades se las denomina *templates* y se caracterizan por no poseer información de estado.

Aunque ambos proyectos se basan en el concepto de nodo virtual, existen algunas diferencias entre ellos.

- En DIADEM la unidad raíz es un procedimiento de biblioteca que proporciona al menos un flujo de ejecución al nodo virtual. De alguna forma el procedimiento raíz puede entenderse como el programa principal del nodo virtual y, por lo tanto, el programa formado por el conjunto de nodos como un conjunto de programas Ada. Por otra parte, pueden existir uno o más paquetes de biblioteca que proporcionan el interfaz (fig. 5.1).

  En YDA un paquete de biblioteca es a la vez unidad raíz y unidad de interfaz. No existe en principio un flujo de ejecución asociado al nodo virtual, excepto el correspondiente a la elaboración e inicialización del paquete raíz. Serán las tareas declaradas en el nodo virtual las que proporcionen los flujos de ejecución necesarios (fig. 5.2).
- La comunicación entre nodos virtuales en DIADEM se realiza mediante llamadas remotas a los puntos de entrada de las tareas declaradas en la parte visible de los paquetes de interfaz.

  En YDA la comunicación se realiza mediante llamadas remotas a los procedimientos declarados en la parte visible del paquete interfaz.

  En ambos casos no está permitido utilizar parámetros de tipo acceso en las invocaciones remotas para evitar la compartición de datos entre nodos virtuales.
- La gestión de la elaboración y de la terminación se realiza de forma diferente en ambos proyectos. En [HUT89] se realiza un análisis de las diferentes soluciones.

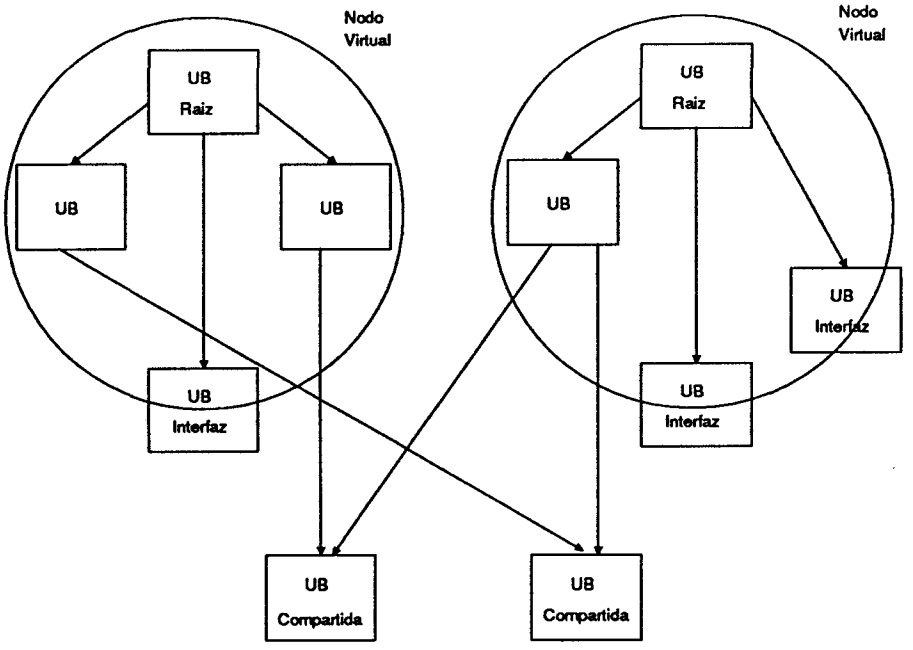

Fig. 5.1 Nodos Virtuales en DIADEM

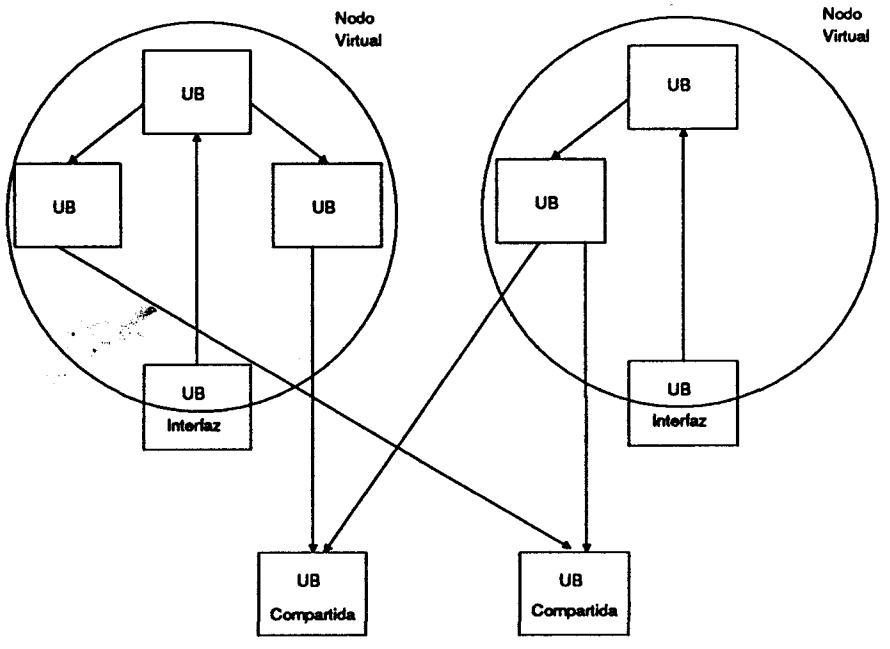

Fig. 5.2 Nodos Virtuales en YDA

- Elaboración del programa. Se debe buscar el máximo paralelismo en la elaboración de los nodos virtuales del programa manteniendo el orden parcial de dependencias existente en el programa. En DIADEM la elaboración de los nodos se realiza en paralelo. Si una llamada llega a una unidad antes

de que ésta se haya elaborado, se rechaza la llamada pudiendo el cliente repetir dicha llamada.

En YDA cuando llega la llamada antes de la elaboración, la llamada se bloquea hasta que ésta se produzca. Con esta estrategia, si el programa no tiene un orden de elaboración debido a dependencias circulares, se produce un interbloqueo en vez de generarse la excepción PROGRAM_ERROR.

- Terminación del programa. La utilización del nodo virtual como unidad de partición permite que la gestión de la terminación de tareas se realice independientemente a nivel de cada partición. La terminación de las tareas dependientes directamente de paquetes de biblioteca, y con ellas la terminación del programa Ada, no está definida en el MRL. No será necesario, por lo tanto, realizar una gestión entre particiones. Así, en DIADEM el programa debe diseñarse de manera que se asegure la correcta terminación del mismo.

  A pesar de la falta de definición del MRL, en YDA un programa termina cuando todas las tareas terminan. El algoritmo para gestionar la terminación del programa se basa en detectar una situación de estabilidad del programa. Una vez detectada, se debe distinguir entre el interbloqueo y la terminación.

Estos dos proyectos han sido de gran interés para adquirir experiencia en la utilización del nodo virtual como unidad de partición. Sin embargo, la utilización plena de este concepto exige su inclusión dentro del lenguaje más que soluciones basadas en "preprocesadores". La necesidad de tipos nodo virtual o el problema en la propagación de excepciones remotas (con una solución preprocesador, si una excepción se propaga a un nodo virtual donde no es visible y de éste se propaga a otro en el cual es visible, el nombre de la excepción se pierde) son ejemplos de esta necesidad.

5.4.3 Inclusión del nodo virtual y otras construcciones similares en Ada

En [VOL90] se recoge una propuesta de incluir el concepto de nodo virtual en Ada. La declaración de un nodo virtual consiste simplemente en la lista de unidades de biblioteca que lo componen. La principal diferencia de esta propuesta es que se permite acceder a todos los paquetes y subprogramas del nodo y no sólo a ciertas unidades de interfaz como en YDA y DIADEM. Además no se restringen las declaraciones que pueden aparecer en la parte visible de los paquetes de biblioteca que forman parte del nodo. Como consecuencia de ello se permite el acceso desde un nodo virtual a variables declaradas en otro nodo.

La máquina virtual resultante de esta propuesta sigue teniendo un modelo de memoria compartida como el original de Ada, frente al modelo mixto definido en los proyectos YDA y DIADEM. La facilidad que introduce esta propuesta sobre el modelo original es que tanto la gestión del *cactus stack* como la terminación de tareas se mantienen internas a cada nodo virtual.

Esta necesidad de incluir el nodo virtual o alguna construcción equivalente en el lenguaje ha sido recogida dentro del proceso de revisión Ada 9X. Antes de analizar las propuestas de Ada 9X se pueden resaltar otras alternativas, entre ellas utilizar los tipos paquete [JES82] o los objetos de un sistema orientado a objetos [GOL90] como nodos virtuales.

5.4.4 Ada 9X y la programación distribuida

Dentro del proceso actual de revisión del lenguaje del cual saldrá el futuro estándar Ada 9X (se espera que la X sea un 3 y por lo tanto Ada 93), se ha recogido la necesidad de modificar el lenguaje para adecuarlo a la programación de sistemas distribuidos. Además de las modificaciones en este campo, existen otras importantes mejoras como la introducción en el lenguaje del modelo de programación orientada a objetos o la inclusión de un mecanismo de sincronización entre tareas más "ligero"

que las citas, los objetos protegidos (*protected records*) [ALV90], que permite satisfacer las exigentes necesidades de los sistemas de tiempo real críticos.

En [Ada9Xa] [Ada9Xb] [BAK90] se describe la solución propuesta en Ada 9X con respecto a la programación distribuida. A continuación se exponen las principales características de esta solución.

Un programa está formado por un conjunto de particiones de dos tipos: Activas y pasivas. Debe existir al menos una partición activa. Cada partición esta formada por un conjunto de unidades de biblioteca.

En cada partición activa uno de los subprogramas se puede designar como subprograma principal y algunos de los paquetes como paquetes interfaz. Esta estrategia es similar a la utilizada en DIADEM pudiéndose interpretar, en cierto sentido, cada partición activa como un programa Ada. A diferencia de DIADEM, las comunicaciones entre particiones se realizan mediante llamadas remotas a subprogramas.

Siguiendo con esta analogía entre partición y programa Ada, en Ada 9X se generaliza el concepto de tarea de entorno que aparecía en MRL. En Ada 83, al flujo de control encargado de la elaboración de las unidades de biblioteca y de invocar al programa principal se le hacía corresponder con una hipotética tarea de entorno. En Ada 9X existirá una tarea de entorno por cada partición activa, la cual se encargará de elaborar las unidades de dicha partición y de invocar al posible subprograma principal.

En Ada 9X se especifica cuando se producen la terminación de las particiones y del programa. Una partición activa termina cuando el subprograma principal (si existe) se completa y todas las tareas dependientes terminan. Un programa termina cuando todas las particiones activas del mismo terminan.

Pueden añadirse y eliminarse particiones activas del programa durante su ejecución. Esta posibilidad facilita la reconfiguración dinámica del programa.

En cuanto a las particiones pasivas, éstas contienen datos visibles a una o varias particiones activas. No tienen asociado ningún flujo de ejecución, ni tarea de

entorno ni tareas declaradas en la partición. Estas particiones sólo pueden estar compuestas de unidades de biblioteca pasivas. Existen diferentes categorías de unidades de biblioteca que pueden identificarse mediante un pragma. Algunas categorías son: Unidad interfaz de llamada remota (proporciona el interfaz de una partición activa), unidad pura (no necesita ser elaborada) y unidad pasiva compartida (no puede contener tareas ni objetos protegidos). Las unidades pertenecientes a estas dos últimas categorías se consideran pasivas y pueden ser componentes de particiones pasivas.

Hay que resaltar que en Ada 9X es responsabilidad de la aplicación proporcionar el subsistema de comunicación. La meta del modelo propuesto es que el entorno de ejecución de cada partición no tenga conocimiento acerca de la distribución del programa, siendo la aplicación responsable de la misma. De esta forma se consigue independencia entre los entornos de ejecución de diferentes particiones y se mantiene un tamaño razonable de entorno de ejecución.

Existen opiniones en contra de esta estrategia que mantienen la necesidad de que el entorno de ejecución sea consciente de la distribución del programa. Un posible ejemplo de esta necesidad es el requisito de que existan identificadores únicos de las excepciones para permitir su propagación remota (este problema aparecía como vimos en las soluciones basadas en preprocesadores). Estos identificadores únicos implicarían que el entorno de ejecución de una partición debe conocer la existencia de otras particiones. Una alternativa es que en vez de propagarse la excepción original se propague una determinada excepción predefinida, por lo tanto conocida, en la partición que la recibe.

La propuesta de Ada 9X, en la que cada partición se comporta casi como un programa Ada, acerca dos estrategias que parecían en principio contrapuestas: Utilizar un único programa Ada frente a utilizar múltiples programas Ada que se comunican entre sí.

En [DOB90] se expone un interesante análisis que apoya este acercamiento entre ambas estrategias. El clásico argumento de que, con la utilización de múltiples

programas, el compilador no puede comprobar las interfaces entre los distintos programas, es rebatido en este artículo.

Estos problemas están asociados al sistema de compilación y no al lenguaje. Es necesario que el entorno de desarrollo soporte el concepto de múltiples programas principales y permita crear simultáneamente los mútiples ejecutables utilizando la misma versión de las unidades comunes. Un ejemplo práctico de esta situación es el compilador de Alsys para sistemas multiprocesadores basados en *transputers* [ALSb].

Existen otras propuestas que, intentando cumplir los mismos requisitos que la propuesta de Ada 9X, plantean soluciones en las que el lenguaje proporciona mayor soporte a la distribución.

El grupo de trabajo sobre nodos virtuales y sistemas distribuidos del cuarto *workshop* de tiempo real [GAR90a] planteó un modelo inicial que posteriormente se ha ido completando [GAR90b].

La propuesta inicial recomienda la inclusión de nuevas construcciones dentro del lenguaje. Algunas de ellas son: Programa, partición, tipo partición y *template* (unidad sin estado interno).

Los tipos partición están asociados implícitamente con tipos acceso anónimos. Esto permite la creación dinámica y la reconfiguración.

En [GAR90b] se proporciona también soporte para la configuración mediante una nueva construcción: El nodo. Un nodo agrupa a un conjunto de particiones, a partir de las cuales el sistema de compilación generará un ejecutable. Los nodos están también asociados implícitamente con tipos accesos anónimos lo que permite su creación dinámica.

Como se puede ver, esta propuesta implica mayores modificaciones en el lenguaje que la correspondiente a Ada 9X. Con ella se pretende que el lenguaje se encargue de la mayoría de los aspectos relacionados con la distribución, frente a Ada 9X donde la aplicación trataba estos aspectos. Las desventajas de esta estrategia es la complejidad resultante en el lenguaje, que complica el diseño del compilador y del

entorno de ejecución. Esta situación podría llevar a una difícil evolución hacia el nuevo Ada. La propuesta Ada 9X, por el contrario, permite una transición más suave.

5.4.5 Implementación del nuevo modelo de Ada en diferentes máquinas

Como se analizó anteriormente el modelo de Ada es bastante adecuado para máquinas con memoria compartida pudiendo utilizarse la tarea como unidad de partición.

La implementación de Ada en máquinas sin memoria compartida es problemática no sólo por el modelo de memoria compartida del lenguaje, sino también por otras características como la gestión de la terminación de las tareas.

Las propuestas revisadas en el apartado anterior definen unidades de partición diferentes de la tarea modificando el modelo de interacción del lenguaje que pasa a ser mixto. Con estas propuestas, además, tanto la gestión de la terminación como la del *cactus stack* se simplifican ya que se realizan dentro de cada partición.

Esta situación puede conducir hipotéticamente a la existencia de dos lenguajes Ada, el original para los sistemas con memoria compartida y el modificado para los sistemas sin memoria compartida, lo cual va directamente contra el objetivo de independencia entre el lenguaje y la máquina subyacente.

Hay que resaltar, sin embargo, que las características del lenguaje modificado, por ejemplo el Ada 9X, son también adecuadas para máquinas con memoria compartida ya que éstas son las menos restrictivas.

Es importante hacer notar que el objetivo de las propuestas de modificación al lenguaje no es únicamente adaptarlo a diferentes tipos de máquinas, sino también modificar la funcionalidad del lenguaje. Así, por ejemplo, entre las características que deben poseer los nodos virtuales, además de las relacionadas con la partición, se encuentran su capacidad de encapsular recursos o su posibilidad de ser unidades de reusabilidad.

Vamos a analizar a continuación la implementación del Ada modificado, tipo Ada 9X o similar, sobre máquinas que sigan los tres modelos de interacción definidos. Antes, sin embargo, vamos a exponer algunas consideraciones.

La nueva máquina virtual Ada presenta un modelo mixto de memoria. Un programa está formado por un conjunto de particiones. Cada partición agrupa a su vez a un conjunto de tareas que comparten memoria.

La asignación de tareas de una misma partición a procesadores que no comparten memoria puede provocar problemas de eficiencia y, además, complica el diseño del entorno de ejecución, que debe proporcionar acceso remoto a datos, gestión de terminación y gestión del *cactus stack* sin el apoyo de la memoria compartida.

Por el contrario, la asignación de tareas de una partición a procesadores que comparten memoria no es problemática. Tanto la compartición de datos como la gestión de la terminación y del *cactus stack* pueden realizarse de forma eficiente, no complicando el entorno de ejecución del lenguaje.

Por lo tanto, la estrategia de configuración consiste en asignar cada partición y sus tareas correspondientes a un grupo de procesadores que compartan memoria.

Esta misma conclusión se puede obtener de la filosofía de Ada 9X. Si una partición es relativamente equivalente a un programa Ada 83 y éste es muy adecuado para ejecutar en máquinas con memoria compartida, la partición de Ada 9X se ejecutará eficientemente en dichas máquinas.

A continuación se analiza la implementación sobre diferentes máquinas.

- Máquinas con memoria compartida y asignación dinámica directa. El modelo mixto de Ada 9X se implementa eficientemente en estas máquinas. En vez de tener una única "bolsa" de tareas ejecutables como en el Ada original, existen varias "bolsas", una por partición. Sin embargo, la situación es equivalente ya que un procesador puede elegir en cada instante una tarea ejecutable de cual-

quier "bolsa". Un procesador ejecutará a lo largo del tiempo tareas de distintas particiones.

- Máquinas con memoria compartida sin asignación dinámica directa. La principal diferencia con el anterior grupo es la necesidad de realizar una asignación en tiempo de compilación (excepto si hay migración de tareas). En esta asignación es interesante asignar tareas de la misma partición a un mismo procesador para obtener la mayor eficiencia. De todas formas, se pueden asignar tareas de la misma partición a diferentes procesadores ya que la existencia de la memoria compartida permite soluciones eficientes sin complicar el entorno de ejecución del lenguaje.

- Máquinas con memoria distribuida. Se realizará una asignación en tiempo de compilación de una o varias particiones a un procesador. El número total de particiones y sus correspondientes tareas debe proporcionar el exceso de paralelismo que permita la utilización eficiente del procesador frente a la latencia y la sincronización. Evitando dividir las particiones entre varios procesadores simplificamos considerablemente la implementación.

- Máquinas con un modelo mixto. En este caso se asignará a cada subconjunto de procesadores con memoria compartida, el suficiente número de particiones para conseguir una utilización eficiente. El conjunto de tareas de las distintas particiones asignadas a un subconjunto de procesadores que comparten memoria forman una "bolsa" común para dicho subconjunto.

Puede ser necesario, sin embargo, asignar tareas de la misma partición a procesadores que no comparten memoria y, por lo tanto, resolver los mismos problemas que con el Ada 83, si se pretende conseguir independencia de la máquina para poder ejecutar un programa de forma eficiente en diferentes configuraciones.

Un programa Ada 9X se mueve entre dos extremos de una gama de posibilidades. En un extremo está el programa con una única partición y bastantes tareas. Es equivalente a un programa Ada 83 y, por lo tanto, define una máquina con memoria

compartida. En el otro extremo está el programa con bastantes particiones y una sola tarea (la correspondiente al subprograma principal) por partición. Se trata de una máquina con un modelo de memoria distribuida. Sería equivalente a tener múltiples programas secuenciales que se comunican, con lo que se pierde toda la funcionalidad del modelo de tareas de Ada.

Los programas formados por pocas particiones con un número relativamente grande de tareas por partición ("cercanos" a Ada 83) pueden resultar problemáticos en entornos con memoria distribuida ya que, si no se dividen las particiones, de ellos se puede obtener poco paralelismo. Para poder aprovechar el paralelismo del programa será necesario asignar tareas de la misma partición a diferentes procesadores, con lo cual el entorno de ejecución debe proporcionar las mismas funciones que en el modelo original de Ada (acceso a datos remotos con utilización de copias locales o accesos de ciclo partido, gestión de la terminación de tareas y del *cactus stack* sin memoria compartida, etc.).

5.5 Implementación de Ada sobre la máquina sistema operativo

En la sección anterior se realizó un estudio general sobre la implementación de Ada, tanto Ada 83 como Ada 9X, en máquinas virtuales genéricas que responden a los tres modos de interacción, a saber, memoria compartida, memoria distribuida y modelo mixto.

Las conclusiones pueden aplicarse tanto a la implementación del lenguaje sobre máquinas desnudas (máquina virtual física) como sobre máquinas con sistema operativo o ejecutivo (máquina virtual sistema operativo).

Sin embargo, existen algunos aspectos específicos de la máquina sistema operativo. En primer lugar, las evaluaciones sobre la eficiencia de la implementación de Ada en una máquina sistema operativo no son directamente aplicables a la máquina física, ya que para poder analizar la utilización del sistema es necesario tener en cuenta tanto la implementación del lenguaje sobre el sistema operativo como la implementación del sistema operativo sobre la máquina física. De esta forma, una im-

plementación poco eficiente en el nivel de sistema operativo puede resultar eficiente globalmente. Por ejemplo, una asignación estática entre **m** tareas Ada y **n** procesos del sistema operativo tal que **m=n,** puede resultar ineficiente, ya que cuando una tarea se bloquea el proceso asociado también queda bloqueado. Sin embargo, si los **n** procesos se asocian a **p** procesadores físicos, tal que **n>p** con suficiente exceso de paralelismo, la solución global puede ser eficiente. Cuando una tarea y su proceso asociado se bloquean, el procesador correspondiente pasa a ejecutar otro proceso, el cual, por su parte, continuará la ejecución de la tarea asociada.

Otro importante aspecto diferenciador es la eficiencia en la gestión de los procesos. Los procesadores de la máquina sistema operativo son virtuales, el entorno de ejecución se encarga de crearlos, eliminarlos y realizar los cambios de contexto correspondientes.

Los sistemas operativos multiusuario no sólo definen una máquina virtual paralela, sino múltiples, permitiendo de esta manera que varios usuarios puedan trabajar simultáneamente. En este tipo de sistemas cada proceso lleva asociada una considerable información de contexto para mantener la seguridad entre usuarios. Este tipo de procesos son "pesados", su gestión implica un tiempo elevado.

La implementación de Ada sobre este tipo de sistemas operativos, proporcionen un modelo de memoria compartida o no, debe realizarse de forma que el número de procesos asociados a un programa sea bajo ya que su gestión es muy costosa. La utilización de un proceso "pesado" por cada tarea resulta de todo impensable.

Si la máquina física es uniprocesador, la solución habitual es asociar todas las tareas a un único proceso como ocurre en todos los compiladores comerciales (correspondencia **m>n, n=1**).

En un multiprocesador se pueden crear unos pocos procesos a los que se asociarán todas las tareas del programa (correspondencia **m>n**). En los compiladores de Encore y Sequent [BEC89] se utiliza esta solución. Cada proceso selecciona una de

las tareas entre las ejecutables y la ejecuta hasta que se queda bloqueada, repitiendo entonces la misma operación.

Existen otros sistemas operativos que proporcionan procesos "ligeros". La gestión de este tipo de procesos es mucho más eficiente que la de los procesos convencionales ya que tienen muy poca información de contexto asociada. Los procesos "ligeros" normalmente comparten memoria. Algunos sistemas operativos monousuario gestionan este tipo de procesos, pero principalmente son los sistemas operativos distribuidos los que soportan este concepto. Este tipo de sistemas, como se expuso en el tercer capítulo, definen una máquina virtual con modelo mixto.

La implementación de Ada sobre este tipo de sistemas operativos puede realizarse asignando todas las tareas del programa a un número considerable de procesos ligeros ya que su gestión es eficiente. Incluso se puede realizar, tanto en uniprocesadores como en multiprocesadores, una asignación de una tarea a un proceso (siempre que éste sea lo suficientemente "ligero"). Esta solución identifica una tarea con un proceso (**m=n**) lo que minimiza, como se vio en el segundo capítulo, el problema de los bloqueos anidados.

Es necesario, además de la existencia de procesos ligeros, que la semántica de los mismos (comunicación entre procesos, planificación, etc.) sea relativamente compatible con la de las tareas Ada, ya que, en caso contrario, la solución puede resultar ineficiente.

Una interesante opción es la implementación de Ada sobre los procesos ligeros de POSIX llamados *pthreads*. El POSIX define un interfaz estándar para el sistema operativo. Además de este estándar básico, POSIX se ocupa de áreas específicas tales como procesamiento de transacciones, seguridad, multiproceso o tiempo real. Dentro de este último campo se ha decidido incluir dentro de POSIX la capacidad para manejar procesos ligeros o *threads* (*pthreads* en POSIX).

Con la inclusión de este concepto la máquina virtual POSIX presenta un modelo mixto. Cada proceso POSIX está compuesto de un conjunto de *pthreads* que comparten memoria.

Se puede asignar cada tarea a un *pthread* pero, para obtener una solución eficiente, es necesario que la funcionalidad de los *pthreads* sea relativamente compatible con la de Ada. En [PAZ90] se analizan las dificultades que existen con el actual modelo de *pthreads*, realizando algunas recomendaciones para aumentar la compatibilidad con Ada.

Hay que resaltar que la implementación de Ada sobre *pthreads* resulta muy interesante ya que la estandarización de POSIX facilitaría considerablemente el desarrollo de compiladores comerciales.

En [CME89] se expone un interesante análisis sobre la implementación de Concurrent C en diferentes entornos, entre otros, un UNIX convencional y un sistema operativo monousuario que proporciona procesos ligeros CTK (*Communications and Tasking Kernel*). En el entorno UNIX todos los procesos del programa se asocian a un único proceso UNIX. En el entorno CTK cada proceso del programa se asocia con un proceso ligero. El cambio de contexto en el primer caso es de 50 μs mientras que en el segundo es de 190 μs. Sin embargo, en la implementación UNIX no se puede aprovechar el paralelismo de la máquina subyacente mientras que en la implementación CTK si que es posible.

La diferencia en el cambio de contexto se debe a que la gestión de las tareas implica, en el último caso, operaciones sobre los procesos ligeros, las cuales a su vez tendrán como consecuencia llamadas al sistema. En el primer caso, sin embargo, la gestión de las tareas no implica operaciones sobre el proceso y, por lo tanto, no son necesarias llamadas al sistema (excepto al principio para crear el proceso y al final para eliminarlo).

Por último, hay que recordar que otro de los factores diferenciadores entre la máquina sistema operativo y la máquina física es la existencia de los bloqueos anida-

dos. Este problema fue analizado de forma genérica en el segundo capítulo y será retomado en el siguiente.

5.6 Sumario del capítulo

El modelo del lenguaje Ada se adapta perfectamente a las características de las máquinas con memoria compartida y asignación dinámica directa.

Las máquinas con memoria compartida pero sin asignación dinámica directa (el código de los procesos no está almacenado en la memoria compartida) también son adecuadas para Ada. En este caso, sin embargo, es necesario realizar una partición estática en tiempo de compilación (con posible migración de tareas en tiempo de ejecución) lo cual no permite una utilización tan eficiente como en el caso anterior.

La utilización de Ada en máquinas sin memoria compartida (memoria distribuida o modelo mixto) es problemática. El acceso a datos remotos puede optimizarse utilizando copias locales o realizando accesos de ciclo partido. Para una utilización eficiente es necesario que el exceso de paralelismo sea suficiente para esconder la latencia en la comunicación. Es necesario también gestionar la terminación de tareas mediante el intercambio de mensajes entre los procesadores. Además de las dificultades para obtener una solución eficiente, se complica considerablemente el diseño del entorno de ejecución que debe implementar accesos remotos y realizar la gestión de la terminación de tareas y del *cactus stack* de manera distribuida.

Frente a estas dificultades se han planteado diversas alternativas basadas en utilizar otras unidades de partición distintas de la tarea. Desde la utilización restringida de unidades de biblioteca hasta el concepto de nodo virtual, existen múltiples alternativas que crean una nueva máquina virtual Ada con un modelo mixto y que simplifican la gestión de la terminación y del *cactus stack.*

El proceso de revisión del lenguaje Ada 9X ha recogido una propuesta relativamente conservadora bastante cercana a la utilización de múltiples programas que se comunican. Esta solución mantiene relativamente sencillo el entorno de ejecución,

dejando a la aplicación la resolución de muchos aspectos relacionados con la distribución. Con esta propuesta se permite una transición suave entre Ada 83 y Ada 9X.

La propuesta Ada 9X define un modelo que se adapta a los tres tipos de máquina. Todas las tareas de una partición se asignarán a un conjunto de procesadores que compartan memoria.

Ada 9X proporciona, por lo tanto, una considerable independencia de la máquina sobre la que se implementa manteniendo la complejidad del lenguaje y de su entorno de ejecución en unos límites razonables. Sin embargo, en algunos entornos con memoria distribuida será necesario repartir las tareas de una partición entre varios procesadores para poder utilizar el paralelismo de máquina, ya que una partición puede contener un número considerable de tareas. En este caso el entorno de ejecución será tan complejo como el correspondiente al Ada 83.

Por último, se han analizado las peculiaridades de la implementación de Ada sobre el sistema operativo. Un aspecto importante en la máquina virtual sistema operativo es la eficiencia en la gestión de los procesos. Según este aspecto se puede diferenciar entre procesos "pesados" (procesos convencionales) y procesos "ligeros" (*threads*).

Sobre el primer tipo de sistemas se asociarán todas las tareas a un único proceso (máquina física uniprocesador) o a unos pocos (máquina física multiprocesador con memoria compartida como el Sequent).

En el caso de procesos ligeros se puede realizar una asociación de una tarea a un proceso ligero (correspondencia **m=n**), siempre que las características de los procesos sean relativamente compatibles con las de las tareas. La utilización de las *pthreads* de POSIX para implementar las tareas Ada es un interesante ejemplo de esta situación. Esta correspondencia es adecuada para minimizar el problema de los bloqueos anidados.

6

Interfaz desde Ada al entorno externo

6.1 Introducción

En este capítulo se retoma el problema de los bloqueos anidados particularizándolo para el caso de Ada.

Este problema surge cuando la acción de una tarea causa que un proceso del sistema operativo se quede bloqueado.

Esta situación ocurre cuando la tarea accede a un objeto fuera del "mundo Ada", por ello, en primer lugar, se describirán los mecanismos que proporciona el lenguaje para acceder a objetos del entorno externo.

A continuación se analizará el problema de los bloqueos en Ada evaluando las repercusiones del mismo dependiendo del tipo de implementación existente. Se propondrán alternativas basadas en utilizar servicios asíncronos para simular los síncronos (bloqueantes).

También se estudiarán otros problemas relacionados como la reentrancia o la posible interferencia entre los servicios utilizados directamente por las tareas y los utilizados por el entorno de ejecución de Ada.

Se expondrán ejemplos prácticos de estos problemas basados en los compiladores Ada de Digital [VAX], Alsys [ALSa], Verdix [VER] y Meridian [MER].

6.2 Mecanismos que permiten el acceso al entorno externo

Existen frecuentes casos en los que es necesario que un programa se relacione con el "mundo exterior" para poder acceder a objetos hardware y software. Algunas situaciones típicas en las que existe esta necesidad son:

- Sistemas empotrados. Es necesario el acceso directo a dispositivos hardware. Para ello el lenguaje debe proporcionar facilidades para direccionar y manipular los registros del dispositivo, tanto en sistemas con mapa común de memoria y de entrada/salida como en sistemas con mapas separados, y mecanismos para gestionar las interrupciones del dispositivo.

- Utilización de software de aplicación no Ada. Ya sea por la necesidad de utilizar software ya existente (propio o comercial), o por la mejor adecuación de otros lenguajes a algunas partes de la aplicación.

- Utilización de los servicios del sistema operativo. Deben existir facilidades para realizar llamadas al sistema o para poder gestionar eventos asíncronos del sistema (señales).

En los capítulos 13 y 14 del MRL se presentan las numerosas características de Ada que permiten el acceso a objetos que no forman parte del "mundo Ada".

De las facilidades expuestas en ambos capítulos, este trabajo se va centrar en las siguientes:

- Gestión de interrupciones (opcional)

- Inserciones de código máquina (opcional)

- Pragma INTERFACE (opcional)

- LOW_LEVEL_IO (opcional)
- Entrada/Salida estándar (obligatorio)

Como se puede ver, el único mecanismo que debe proporcionar obligatoriamente una determinada implementación es la E/S estándar. Esta implementación proporcionaría la funcionalidad mínima quedando el programa bastante "aislado" del entorno externo. Unicamente se podría acceder al sistema de ficheros. La figura 6.1 representa gráficamente esta situación.

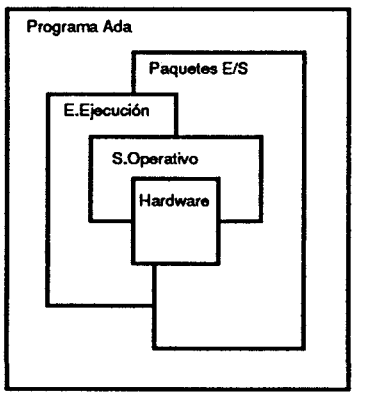

Fig. 6.1 Sistema Ada con el mínimo interfaz

Sin embargo, en muchas ocasiones, es necesario tener un mayor acceso al entorno externo (fig. 6.2). En sistemas que sólo incluyan la E/S estándar, el acceso a un determinado objeto puede conseguirse mediante paquetes de interfaz proporcionados por la implementación, pero se trata de una solución parcial. Para obtener una buena

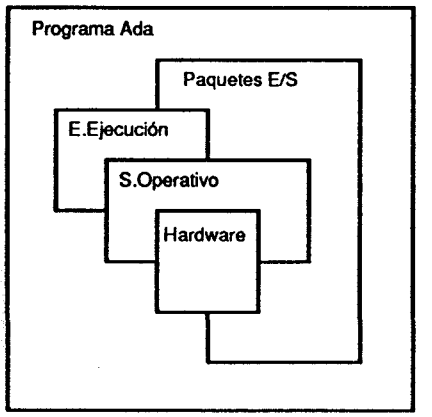

Fig. 6.2 Sistema Ada con acceso al S.O. y al hardware

funcionalidad es importante que un determinado compilador soporte la mayoría de las facilidades opcionales de Ada.

6.2.1 Gestión de interrupciones

Una de las necesidades básicas de un lenguaje para tiempo real es que proporcione mecanismos para el manejo de las interrupciones de los dispositivos. Ada, por su parte, permite asociar una interrupción con un punto de entrada mediante el uso de una cláusula de representación.

Cuando se produce la interrupción se realiza una llamada a dicho punto de entrada. La cita se ejecuta con mayor prioridad que cualquier tarea del programa. No pueden definirse diferentes prioridades para distintas interrupciones, ni tareas con mayor prioridad que ninguna interrupción (esto posibilita que la tarea manejadora de una interrupción de alta prioridad pueda quedar bloqueada por una interrupción de baja).

La mayoría de las características de este mecanismo son dependientes de la implementación (p.ej. la posibilidad de las llamadas condicionales y temporizadas asociadas a la interrupción).

Con los puntos de entrada de una tarea, además de interrupciones hardware, se pueden asociar otros eventos como señales del sistema operativo. El VAX-Ada permite la asociación de un AST (*Asynchronous System Traps*) con un punto de entrada.

6.2.2 Inserciones de código máquina

Este mecanismo proporciona una forma controlada de insertar código máquina en un programa Ada de forma que éste quede claramente aislado del resto. Para ello el lenguaje impone las siguientes restricciones sobre su utilización:

- Sólo se permite insertar código máquina dentro del cuerpo de un procedimiento.

En la parte declarativa de un procedimiento que incluya código máquina sólo pueden aparecer cláusulas **use** y pragmas. No se permiten declarar tipos ni variables locales.

- Las únicas sentencias que pueden aparecer en el cuerpo de un procedimiento con código máquina son sentencias de código máquina.

- Un procedimiento con código máquina no puede declarar un manejador de excepciones.

Cada instrucción de código máquina aparece en el programa como una sentencia de código (*code statement*), que consiste en un agregado de registro que define la instrucción correspondiente. El tipo del registro debe estar predefinido en el paquete de biblioteca MACHINE_CODE.

Los detalles y características de este mecanismo son, como es lógico, muy dependientes de la implementación. Una determinada implementación puede restringir ciertos usos del mismo, o definir pragmas dependientes de la implementación para especificar las convenciones en las llamadas a procedimientos o en el uso de registros de la máquina. El compilador de XD y el compilador de Verdix permiten inserciones de código máquina.

Este mecanismo debe ser usado sólo cuando sea estrictamente necesario. Por ejemplo, puede necesitarse para optimizar secciones críticas de un programa cuyas restricciones temporales no pueden satisfacerse con el código generado por el compilador. También puede ser necesario para acceder a objetos hardware directamente.

6.2.3 Pragma INTERFACE

Este mecanismo permite que un subprograma escrito en otro lenguaje pueda invocarse desde un programa Ada. Este pragma asocia la especificación de un subprograma Ada con una rutina escrita en otro lenguaje. Dicha rutina no se incluye en el fuente del programa llevándose a cabo la asociación en tiempo de montaje. Estas características son bastante diferentes de las correspondientes a las inserciones de

código máquina, las cuales se incluían en el fuente y, por lo tanto, se resolvían en tiempo de compilación.

La utilización de este pragma presenta aspectos dependientes de la implementación pudiendo existir pragmas no estándar para especificar dichos aspectos. Así, en Verdix el pragma INTERFACE_NAME permite especificar el nombre de la rutina con la que se asocia el subprograma Ada (el nombre de la rutina puede no ser un identificador válido en Ada). En VAX-Ada, por su parte, existen pragmas para especificar los mecanismos utilizados en el paso de parámetros. El pragma IMPORT_VALUED_PROCEDURE de VAX-Ada permite llamar a funciones en las que no todos los parámetros son exclusivamente de entrada (en Ada las funciones sólo pueden tener parámetros de entrada).

En Verdix también existen pragmas que permiten acceder desde el programa a objetos definidos en otros lenguajes (pragma INTERFACE_OBJECT). Este tipo de pragmas se recoge también en la propuesta Ada 9X.

El pragma INTERFACE se utiliza normalmente para acceder a software ya existente, por ejemplo bibliotecas de aplicación comerciales (sistemas de gestión de base de datos, paquetes gráficos, etc.).

6.2.4 LOW_LEVEL_IO

Es un paquete estándar que permite controlar dispositivos físicos. Incluye dos operaciones SEND_CONTROL y RECEIVE_CONTROL. Este mecanismo proporciona mayor modularidad que la alternativa de asociar variables con los registros del dispositivo mediante cláusulas de representación. Aunque en este último caso se pueden encapsular todos los accesos a un dispositivo en un paquete que gestione dicho dispositivo.

6.2.5 Entrada/Salida estándar

Es el único mecanismo para acceder al mundo externo que es de carácter obligatorio. Proporciona operaciones orientadas a ficheros y, por lo tanto, permite acceder a objetos cuyo comportamiento sea de ese tipo.

6.3 Los bloqueos anidados en Ada

Como se expuso en el segundo capítulo los bloqueos anidados se producen cuando, como consecuencia de la ejecución de un procesador virtual, se realiza una interacción con el entorno de ejecución de un nivel inferior que bloquea el correspondiente procesador del nivel inferior disminuyendo el paralelismo de la máquina.

Si trasladamos este problema a Ada en su implementación sobre el sistema operativo, el bloqueo anidado tendrá lugar cuando la ejecución de una tarea genera de alguna forma una llamada bloqueante al sistema. El proceso del sistema correspondiente quedará bloqueado afectando al paralelismo de la máquina.

Este problema puede presentarse en varias situaciones. Por un lado, los paquetes correspondientes a la entrada/salida estándar y a la entrada/salida de bajo nivel pueden incluir llamadas bloqueantes al sistema.

Por otro lado, la utilización del pragma INTERFACE y de las inserciones de código máquina pueden producir directa o indirectamente (a través de una biblioteca de aplicación) llamadas bloqueantes al sistema.

La repercusión de este problema fue estudiada de una forma genérica en el segundo capítulo. A continuación aplicaremos las conclusiones alcanzadas sobre diferentes ejemplos de implementación de Ada sobre el sistema operativo.

En máquinas físicas monoprocesador con un sistema operativo que proporcione procesos convencionales ("pesados"), todas las tareas se asocian a un único proceso (correspondencia **m>n, n=1**). Cuando una tarea realiza directa o indirectamente una llamada bloqueante al sistema, todo el programa queda bloqueado.

En máquinas físicas con múltiples procesadores con asignación dinámica directa y un sistema operativo con procesos convencionales, las tareas del programa se asocian a un número limitado de procesos (correspondencia **m>n, n>1**) como en los compiladores Ada para máquinas Sequent y Encore. Cuando una tarea invoca directa o indirectamente una llamada bloqueante el proceso asociado se bloquea degradando el paralelismo del sistema ya que, a partir de ese momento, habrá un proceso menos para ejecutar las tareas del programa.

Cuando el sistema operativo proporciona procesos ligeros, ya sea sobre una máquina física monoprocesador o multiprocesador, puede asociarse cada tarea a un proceso ligero (correspondencia **m=n**). De esta forma, cuando se bloquea el proceso únicamente se queda bloqueada la tarea que realizó la llamada al sistema, lo cual es la situación ideal.

Independientemente del tipo de correspondencia entre las tareas y los procesos, una alternativa es evitar la utilización de servicios bloqueantes (síncronos) y en su lugar utilizar servicios no bloqueantes (asíncronos) que notifiquen su terminación mediante algún tipo de señal que puede asociarse a un punto de entrada. En VAX-Ada, por ejemplo, en vez de utilizar la versión síncrona del servicio del sistema para realizar una operación de entrada/salida (QIOW), la cual bloquearía el programa, se puede usar el servicio asíncrono (QIO), de forma que notifique la terminación del servicio mediante un AST que estará asociado a un punto de entrada donde la tarea se quedará bloqueada esperando.

El programador puede utilizar esta alternativa cuando quiere invocar un servicio del sistema. En cuando a los paquetes predefinidos tales como los correspondientes a la entrada/salida estándar o a la de bajo nivel, pueden diseñarse de forma que utilicen servicios asíncronos y llamadas explícitas al entorno de ejecución Ada. Cuando una tarea invoca una operación de estos paquetes que implique una llamada al sistema, se usará dentro del paquete la versión asíncrona de la llamada y, a continuación, se le notificará al entorno de ejecución que la tarea está suspendida. El

entorno de ejecución hará que el proceso asociado a la tarea suspendida pase a ejecutar otra tarea que esté en estado ejecutable.

En VAX-Ada la entrada/salida sigue el comportamiento descrito, proporciona operaciones asíncronas con respecto a los procesos pero síncronas respecto a las tareas. Se incluye también el paquete TASKING_SERVICES que permite acceder a los servicios del sistema operativo VMS pero bloqueando, cuando sea necesario, únicamente la tarea que invocó el servicio.

Las soluciones expuestas pueden introducir conflictos semánticos con algunas definiciones incluidas en el MRL. Según Fantechi [FAN84], sería necesario ampliar el concepto de punto de sincronización para que incluya factores externos que causan la suspensión de la tarea, como es el caso de la invocación de las llamadas bloqueantes al sistema (si se realizan accesos de ciclo partido a las variables compartidas, esta operación sería también un punto de sincronización).

Si no se consideran estos factores externos como puntos de sincronización, la tarea correspondiente, según la definición del lenguaje, seguiría en estado ejecutable aunque la implementación la mantenga suspendida y, por lo tanto, podría no cumplirse la regla de prioridades de Ada.

En el artículo se propone incluir en el MRL el concepto de puntos de sincronización externos. Como se trata de una característica dependiente de la implementación, cada compilador debería incluir en el apéndice F de su documentación los puntos de sincronización externos que proporciona. Hay que recordar que el MRL especifica que los aspectos dependientes de la implementación se expondrán en dicho apéndice.

6.4 Otros problemas relacionados

Para conseguir el máximo paralelismo del sistema es necesario que exista reentrancia en el acceso a los objetos. En caso contrario los accesos deben realizarse secuencialmente disminuyendo el paralelismo del sistema.

Es importante que, tanto el entorno de ejecución como el sistema operativo, sean reentrantes para permitir que se realicen múltiples accesos simultáneamente. Las implementaciones convencionales de UNIX, por ejemplo, no permiten esta reentrancia. Sin embargo, la implementación de UNIX sobre la que se apoya el compilador de Ada para las máquinas Encore es reentrante.

En general, existen objetos software que no son reentrantes. Las rutinas escritas en FORTRAN, por ejemplo, no son normalmente reentrantes. El compilador Meridian para DOS también restringe la ejecución de operaciones de entrada/salida debido a que BIOS no es reentrante.

Por lo tanto, cuando se accede a objetos no reentrantes, sea el sistema operativo o software de aplicación, es necesario que el entorno de ejecución o el programador explícitamente, incluyan mecanismos de sincronización que eviten los accesos simultáneos. Esta sincronización es necesaria incluso en entornos monoprocesadores, debido al carácter expulsivo de la planificación en Ada y a la posibilidad de la utilización de rodajas de tiempo entre las tareas de igual prioridad (pragma TIME_SLICE en VAX-Ada).

Otro problema que aparece cuando desde un programa Ada se invoca directamente servicios del sistema operativo, es la posible interferencia de estos servicios con los utilizados por el entorno de ejecución. Esta interferencia puede llevar a un funcionamiento incorrecto del programa pudiéndose producir errores de ejecución, un bloqueo del programa o un comportamiento anómalo del mismo.

En VAX-Ada puede haber conflictos si el programa utiliza directamente los servicios del VMS para manejar AST, ya que el entorno de ejecución utiliza estos servicios para implementar algunas funciones del lenguaje, tales como la sentencia **delay**, las sentencias de entrada/salida o algunas operaciones entre tareas.

En el compilador de Alsys sobre UNIX se recomienda no modificar los manejadores de ciertas señales ya que éstas son utilizadas por el entorno de ejecución, el cual habrá establecido previamente sus propios manejadores. El entorno de ejecución

utiliza **alarm** para implementar la sentencia **delay**, las llamadas temporizadas y las rodajas de tiempo. Otro grupo de señales (SIGILL, SIGFPE, etc.) se utilizan para la gestión de las excepciones.

6.5 Sumario del capítulo

En este capítulo se han estudiado los mecanismos que proporciona Ada para que un programa pueda interactuar con el entorno externo, y se ha analizado la problemática asociada a la utilización de los mismos en ciertas situaciones.

Uno de los problemas identificados es el ya conocido problema de los bloqueos anidados. Se ha evaluado su repercusión en diferentes implementaciones de Ada sobre el sistema operativo.

En las implementaciones convencionales en un monoprocesador, el programa entero resulta bloqueado. En multiprocesadores con procesos convencionales, como en las máquinas Encore, disminuye el paralelismo del sistema. La situación ideal se produce cuando existen procesos ligeros y existe una correspondencia directa entre tarea y proceso, ya que en este caso sólo se bloquea la tarea correspondiente.

Se ha analizado también la utilización de servicios asíncronos (no bloqueantes) para simular los síncronos (bloqueantes). Algunos compiladores comerciales siguen esta alternativa. En VAX-Ada la entrada/salida estándar es síncrona a nivel de tarea pero asíncrona a nivel de proceso y, además, existe un paquete TASKING_SERVICES que permite acceder a los servicios del sistema bloqueando únicamente la tarea correspondiente.

Se han identificado otros problemas en el acceso al entorno externo. Por un lado, problemas de reentrancia en el acceso a ciertos objetos y, por otro, la posible interferencia entre los servicios del sistema que puede utilizar una tarea al acceder a un objeto y los servicios que usa el entorno de ejecución.

7

Acceso a GKS desde un programa concurrente Ada

7.1 Introducción

En este capítulo se analiza la utilización del estándar **Graphical Kernel System (GKS)** desde un programa Ada. El interés de este ejemplo reside en que presenta algunos de los problemas analizados en el capítulo anterior. Por un lado, ciertas operaciones del GKS implican llamadas bloqueantes al sistema operativo y, por otro, existen problemas de reentrancia en el acceso al mismo.

El GKS es un estándar internacional propuesto por el Grupo de Trabajo sobre Gráficos de ISO con el código IS7942 [GKS85] [HOP87]. Se han definido también los *bindings* de GKS con distintos lenguajes, entre ellos el de Ada cuyo código es IS8651-3 [GKS88]. Un ejemplo de construcción del *binding* Ada a partir de un *binding* C, utilizando el pragma INTERFACE y cláusulas de representación, se puede encontrar en [VEG90].

El análisis contenido en este capítulo deriva del proyecto SEISAE. El objetivo de este proyecto era construir una aplicación de representación cartográfica con un interfaz gráfico interactivo. En el proyecto se seleccionó Ada y GKS por su adecuación a las necesidades del proyecto, así como por el carácter estándar de ambas

herramientas. Se trata de un proyecto de un volumen considerable (aproximadamente 60.000 líneas de código fuente Ada), pudiéndose encontrar una descripción del diseño y las características de la solución adoptada en [CAR88].

Uno de los requisitos identificados en el sistema fue la necesidad de que las tareas accediesen concurrentemente de manera independiente a GKS. En nuestra opinión esta necesidad se debe presentar en bastantes aplicaciones muy diferentes que utilicen Ada y GKS. Esto es debido a que GKS se encarga de controlar múltiples dispositivos gráficos y parece natural la utilización de una tarea por cada dispositivo, lo que implica el acceso concurrente a GKS. Las características de GKS, sin embargo, dificultan el cumplimiento de este requisito.

En este capítulo se examinará, en primer lugar, la problemática asociada al acceso concurrente a GKS desde Ada. A continuación, se proponen y evalúan varias soluciones para, por último, describir detalladamente la implementación de la solución adoptada. Esta solución se basa en un planificador que controla el acceso de las tareas a GKS eliminando los problemas que conllevan las operaciones bloqueantes.

La mayor parte del análisis que se expondrá a continuación está recogida en [PER89] y [ZAM90]. En el primer artículo se realiza una exposición con un punto de vista más centrado en Ada, mientras que en el segundo se describe una solución más independiente del lenguaje utilizado. En este capítulo, sin embargo, se realiza un análisis algo diferente, enmarcando este ejemplo dentro de la problemática general estudiada en el capítulo anterior. Además, se presenta una solución más completa profundizando en los aspectos de la gestión de la entrada y sus repercusiones en el diseño del planificador.

7.2 Problemas de acceso a GKS

Las primitivas de GKS no son autocontenidas, esto es, no incluyen toda la información necesaria para ejecutarlas. El resultado de una primitiva depende, además de los parámetros de la misma, del estado en el que se encuentre GKS en ese instante. A dicho estado le llamaremos "estado gráfico global".

Si se necesita que un conjunto de tareas accedan simultáneamente de forma independiente a GKS, es necesario que exista un estado gráfico para cada tarea. En este caso no es suficiente con serializar los accesos al GKS, como con los problemas normales de reentrancia, sino que también se necesita asegurar que cuando una tarea accede a GKS el estado gráfico global coincida con el estado gráfico de la tarea. En caso contrario, el efecto de ejecutar una primitiva por parte de una tarea quedará determinado por el estado gráfico de otra tarea.

El estado gráfico global incluye la siguiente información:

- El estado de operación del GKS
- Los atributos gráficos, ya sean individuales o índices *bundle*
- Las *workstations* que estén activas
- La transformación de normalización seleccionada
- El segmento que esté abierto (si existe alguno)
- El estado de *clipping*
- El procedimiento de manejo de errores seleccionado

El estado de operación de GKS permite controlar las primitivas que se pueden invocar en cada momento. Así, si se ejecuta la primitiva GKS_CLOSE_SEGMENT cuando el estado de operación no es el correspondiente a que existe un segmento abierto, se produciría un error. Hay que asegurar, por lo tanto, que cuando una tarea invoca una primitiva, el estado de operación es el adecuado, o sea, el correspondiente a la última primitiva GKS ejecutada por la tarea, con independencia de que desde entonces otras tareas hayan ejecutado otras primitivas.

De la misma forma, cada tarea selecciona una transformación de normalización, unos atributos gráficos, un estado de *clipping* y, opcionalmente, un procedi-

miento de manejo de errores determinado, los cuales deben de estar seleccionados cuando la tarea ejecuta una primitiva.

El problema con las *workstations* (dispositivo gráfico del GKS) activas es similar. En GKS una primitiva de salida se refleja en todas las *workstations* que estén activas en ese instante. Es preciso asegurar que, cuando una tarea llama a una primitiva GKS, sólo las *workstations* usadas por dicha tarea estén activas.

En cuanto a los segmentos (entidad que permite almacenar información gráfica), hay que asegurar que no se incluyan primitivas ejecutadas por una tarea en un segmento asociado a otra. Debido a las características de los segmentos, el cumplimiento de este requisito es relativamente complejo.

Los problemas relacionados con el estado gráfico global pueden considerarse como problemas de reentrancia con ciertas características específicas. En la figura 7.1 se muestra un ejemplo en el que la tarea 1 utiliza la *workstation* ws_1 mientras que la tarea 2 ejecuta sobre la *workstation* ws_2. Se ha representado en la figura una posible ejecución de ambas tareas con los correspondientes problemas. En [A] el segmento

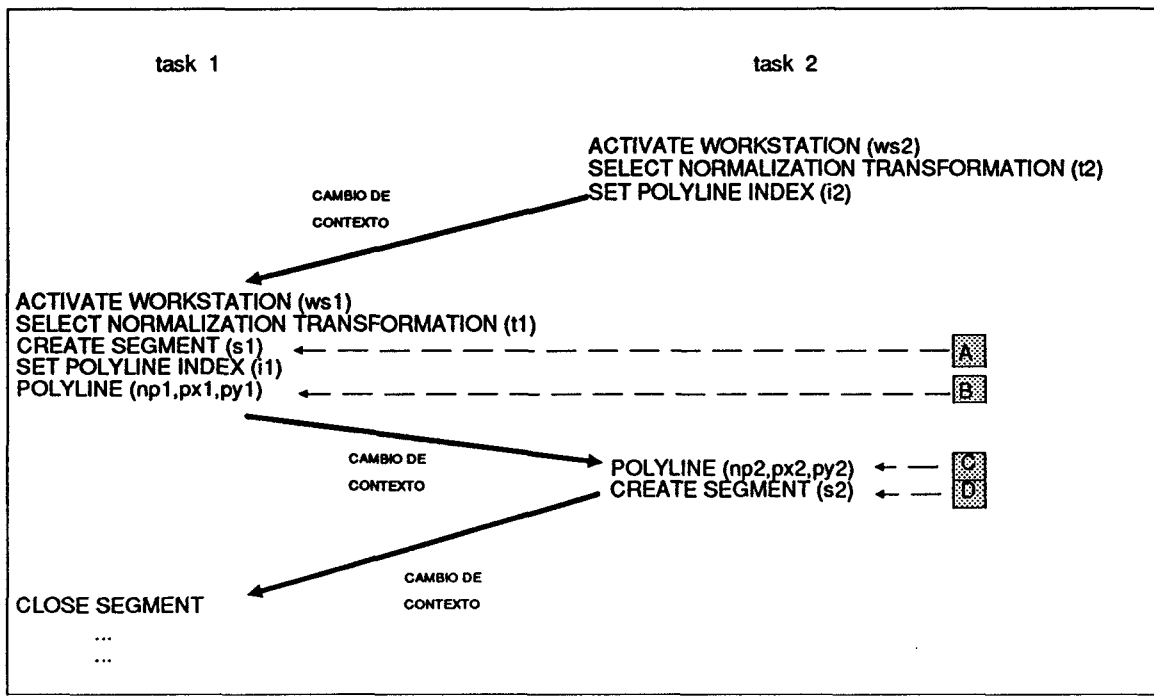

Fig. 7.1 Ejemplo de ejecución conflictiva

se almacenará en una *workstation* no deseada. En [B] una polilínea ejecutada por la tarea 1 aparecerá en ws_2. En [C] la polilínea de la tarea 2 será almacenada en el segmento s_1, utilizando el índice i_1 y la transformación t_1. Por último, en [D] se produciría un error debido a que ya existe un segmento abierto.

Además de estos problemas de reentrancia, existen en GKS problemas debido a la existencia de llamadas bloqueantes. Estos problemas se presentan con la entrada gráfica de GKS.

En GKS existen tres modos de entrada: REQUEST, SAMPLE Y EVENT.

- Modo REQUEST. Se trata de una forma de entrada bloqueante (síncrona). Cuando se ejecuta una primitiva de entrada de este modo, el proceso queda bloqueado hasta que el usuario introduce una entrada por el dispositivo correspondiente. La repercusión de esta situación depende, como se expuso anteriormente, del tipo de implementación de Ada. Si todas las tareas están asociadas a un único proceso, que es el caso más habitual en monoprocesadores, el programa entero quedará bloqueado. Si las tareas están asociadas a varios procesos, disminuirá el paralelismo de la máquina, que dispondrá de un proceso menos. Si existe una correspondencia directa entre tareas y procesos (ligeros evidentemente), el bloqueo sólo afectará a la tarea que invocó la primitiva de entrada, lo cual es el comportamiento ideal.
  Se analizará en los siguientes apartados la utilización de operaciones asíncronas para evitar los bloqueos que ocurren en los dos primeros casos.

- Modo SAMPLE. Es un modo de entrada no bloqueante y, por lo tanto, no causa ningún problema.

- Modo EVENT. Permite obtener entrada de múltiples dispositivos simultáneamente. GKS mantiene una cola FIFO donde se almacenan las entradas de dichos dispositivos. Sería necesario asegurar que cada tarea obtiene únicamente entradas de los dispositivos que está manejando. Se trata, por lo tanto, de un problema similar a los correspondientes al estado gráfico global.

Existe también otro problema con la entrada EVENT. En la primitiva que permite recoger la primera entrada almacenada en la cola FIFO, se incluye un parámetro que establece un plazo de tiempo durante el cual el proceso quedará bloqueado si la cola está vacía. Este bloqueo temporal afectará en mayor o menor grado, dependiendo del tipo de implementación, como ocurría con el modo REQUEST (al fin y al cabo el modo REQUEST puede entenderse como un modo EVENT con plazo infinito). Se analizará también en este caso, la utilización de operaciones asíncronas para eliminar estos problemas.

7.3 Algunas posibles soluciones

Una solución a todos los problemas expuestos debería, por un lado, proporcionar acceso exclusivo a GKS estableciendo en cada momento el estado gráfico de la tarea que está invocando la primitiva. Por otro lado, se debería simular la entrada bloqueante a partir de operaciones no bloqueantes.

7.3.1 Mantenimiento del estado gráfico de cada tarea

En el anexo D al estándar que describe el *binding* de Ada para GKS (el anexo no forma realmente parte del estándar, proporcionando únicamente una información complementaria) se trata brevemente el tema de la utilización de GKS desde un programa con múltiples tareas. En dicho anexo se propone, para evitar los problemas de reentrancia, definir una tarea con un punto de entrada por cada primitiva GKS. De esta forma se asegura el acceso exclusivo a GKS.

Como se expuso en el apartado anterior, esta solución no es suficiente para eliminar los problemas que surgen con el tipo de utilización que se persigue (múltiples tareas accediendo a GKS de una forma independiente).

En esta situación será necesario que cuando una tarea ejecuta una primitiva, el estado gráfico sea el correspondiente a dicha tarea. Esto requiere que el estado gráfico de una tarea sea salvado después de ejecutar una primitiva, para poder restaurarlo posteriormente cuando la tarea invoque otra primitiva.

Se podría modificar la solución propuesta en el anexo D para que pudiese mantener el estado gráfico de cada tarea. Estas modificaciones serían, entre otras, las siguientes: Cada primitiva debe incluir un parámetro que especifique la tarea que la invoca (debería existir un mecanismo para asignar y liberar estos "identificadores" de tarea); debe existir una tabla para almacenar el estado gráfico de las diferentes tareas; antes de invocar una primitiva se debe restaurar el estado gráfico de la tarea que estará almacenado en la tabla; después de invocar una primitiva se debe salvar el estado gráfico en la entrada correspondiente de la tabla.

La operación de salvar el contexto gráfico implicaría las siguientes acciones:

- Salvar los valores actuales de los atributos gráficos, de la transformación de normalización, del estado de operación y del estado de *clipping*, utilizando las funciones INQUIRE para obtenerlos.

- En el caso de que existan *workstations* activas o un segmento abierto, además de almacenar sus identificadores en la entrada correspondiente de la tabla, es necesario desactivar las *workstations* y cerrar el segmento.

La restauración del estado gráfico de una tarea implicaría acceder a la tabla para recuperar dicho estado y, a continuación, ejecutar las primitivas correspondientes para volver a establecerlo. Habría que ejecutar las siguientes acciones:

- Seleccionar los valores adecuados para los atributos gráficos, la transformación de normalización y el estado de *clipping*.

- Activar las *workstations* correspondientes.

- Si existía un segmento abierto cuando se salvó el estado, es preciso restaurar el segmento para que siga en el mismo estado en el que quedó.

Segmentos

Las características de la gestión de los segmentos dificultan considerablemente la restauración de los mismos. Por un lado, sólo puede existir un segmento abierto en cada instante y, por otro lado, una vez que se cierra un segmento no puede volver a ser abierto.

Cuando se salva el estado de una tarea es necesario cerrar el segmento ya que, en caso contrario, cualquier operación realizada por otra tarea se almacenaría en el mismo. La imposibilidad de reabrir el segmento impide, en principio, restaurarlo para que la tarea que lo creó pueda seguir incluyendo en él, de forma transparente, otras primitivas gráficas.

Para resolver este problema se han planteado dos soluciones basadas en la utilización de una *workstation* de tipo WISS, la cual debe estar activa durante la creación del segmento. Una *workstation* WISS sirve como almacenamiento de segmentos, sin estar asociada a ningún dispositivo gráfico. GKS permite que un segmento almacenado en una *workstation* WISS sea copiado a una *workstation* normal.

La primera solución restaura el segmento tal y como estaba cuando la tarea ejecutó la última primitiva. Para ello crea un segmento con los mismos atributos (visibilidad, prioridad, etc.) que el original e inserta en él el segmento almacenado en el WISS. Esta acción requeriría un renombrado previo del segmento original cerrado para permitir que el nuevo segmento mantenga el mismo identificador que el antiguo, y un posterior borrado del segmento original renombrado, el cual ya no es necesario.

Esta solución es general puesto que el segmento se ha recuperado de una forma transparente a la tarea, la cual encontrará el mismo segmento con los mismos atributos, sin embargo no es muy eficiente. Cada vez que se restaura un segmento abierto hay que copiar desde el WISS todas las primitivas almacenadas en el segmento desde su apertura. De esta manera, si **k** es el número de veces que se restaura el segmento

durante su creación, el tiempo invertido en la restauración de los segmentos será del orden de $\mathbf{k}^2$.

La segunda solución se basa en crear un nuevo segmento con los mismos atributos, en vez de restaurar el original. De esta forma, se creará un nuevo segmento por cada restauración, el cual contendrá un fragmento del segmento original. Cuando se cierra el segmento, se creará de nuevo el segmento original insertando en él desde el WISS cada uno de los segmentos que contienen los fragmentos. Esta solución es más eficiente puesto que cada primitiva almacenada en el segmento sólo se copia una vez desde el WISS, al cerrar el segmento, lo cual implica un tiempo del orden de k. Sin embargo, esta solución es menos general y presenta algunos problemas, ya que durante el proceso de creación del segmento, los cambios de atributos y transformaciones sobre el segmento hay que aplicarlos sobre todos los fragmentos.

Ambas soluciones presentan restricciones debido a que utilizan la primitiva INSERT SEG para copiar los segmentos desde el WISS. Esta primitiva no utiliza las *viewports* almacenadas en el segmento sino la actual, por lo tanto no podrán gestionarse segmentos que utilicen más de una *viewport*. Este tipo de segmentos deberían ser explícitamente fragmentados por el programador.

7.4 Gestión de la entrada

Como se expuso antes, existen problemas de bloqueo con el modo de entrada REQUEST y con el modo EVENT cuando se recoge una entrada de la cola FIFO (mediante la primitiva GKS_AWAIT_EVENT) especificando un plazo de espera. Existían problemas también debido a que GKS mantiene una única cola FIFO.

La solución propuesta a estos problemas tiene las siguientes características:

- No se utiliza el modo REQUEST simulándolo a partir del modo EVENT. También se simula el plazo de espera correspondiente a la primitiva que permite recoger una entrada de la cola. Por lo tanto, se simulan servicios bloqueantes a partir de otros no bloqueantes.

- Cuando una tarea realiza una petición de entrada en modo REQUEST, se pone el dispositivo correspondiente en modo EVENT y se bloquea la tarea.

- Por cada dispositivo en modo EVENT o que tenga pendiente una petición de entrada en modo REQUEST, se almacenará el identificador del dispositivo y la tarea asociada, ya sea la que puso el dispositivo en modo EVENT o la que inició la petición en modo REQUEST.

- En GKS no existe ninguna operación de entrada puramente asíncrona. Con el modo EVENT se pueden recoger los eventos de la cola sin bloquearse (para ello el plazo debe ser igual a cero) pero, al no generarse ningún tipo de señal cuando llega una entrada, es necesario realizar un sondeo (*polling*) periódico para detectar la llegada de una entrada. Cuando ocurre esto, se consultará en la información almacenada sobre los dispositivos para decidir a qué tarea corresponde y si se trata, a nivel de la aplicación, de una entrada REQUEST o EVENT.
  Si se trata de una entrada REQUEST, se pondrá el dispositivo en modo REQUEST, se borrarán otras posibles entradas en la cola correspondiente a dicho dispositivo (con el modo REQUEST se obtiene una sola entrada), se desbloquea la tarea y se le devuelve la entrada leída.
  Cuando se trata de una entrada para un dispositivo en modo EVENT, se almacenará dicha entrada en la cola de la tarea correspondiente. Existirá una cola por cada dispositivo en modo EVENT.

- Las primitivas que recogen el primer elemento de la cola accederán a la cola particular de la tarea correspondiente. Si estas primitivas llevan asociado un plazo, y cuando se ejecutan no existe ninguna entrada en la cola de la tarea, se bloquea la tarea iniciándose la temporización (existirá una tarea de alta prioridad que se encargue de gestionar las temporizaciones). Si llega una entrada para esa tarea, se detiene la temporización, se desbloquea la tarea y se le entrega dicha entrada. Si, por el contrario, se cumple el plazo, se desbloquea la tarea y se la informa de que no existe ninguna entrada para ella.

7.5 Implementación de la solución resultante

La solución propuesta hasta ahora, con una tarea que tiene un punto de entrada por cada primitiva GKS y que se encarga de salvar y restaurar el estado gráfico de las tareas, no es eficiente. Con dicha solución será preciso salvar y restaurar el estado por cada primitiva, lo cual es claramente ineficiente debido a que estas operaciones son bastante costosas, sobre todo si existe un segmento abierto. Se puede optimizar la solución evitando salvar y restaurar el estado cuando una misma tarea ejecuta consecutivamente dos primitivas GKS. Sin embargo, una solución eficiente debe permitir que una tarea ejecute sin interrupción un conjunto suficiente de primitivas para compensar el *overhead* de salvar y restaurar el estado. Haciendo una analogía con un sistema operativo multiproceso, es más eficiente ejecutar un conjunto de instrucciones de cada proceso que ejecutar de forma intercalada instrucciones de los distintos procesos.

La solución definitiva sigue esta filosofía. Para ello incluye un planificador que deja acceder a GKS exclusivamente a una tarea, hasta que se cumplen ciertas condiciones, momento en el cual el planificador realiza un cambio de contexto salvando el estado gráfico de la tarea expulsada y restaurando el de la nueva tarea.

Siguiendo la analogía con un sistema operativo, cada tarea que accede a GKS puede estar en cuatro estados (fig. 7.2):

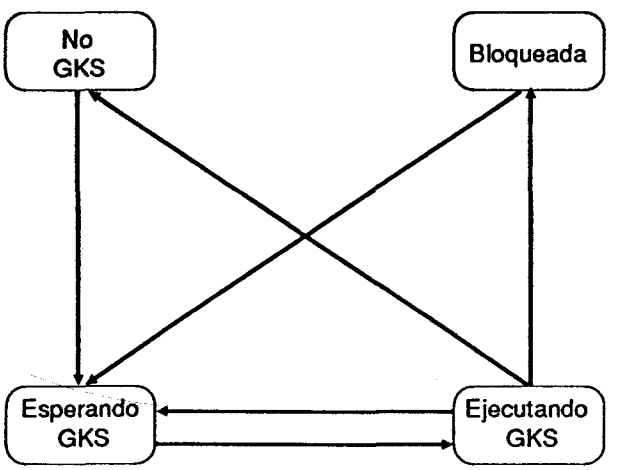

Fig. 7.2 Diagrama de estado de una tarea

- No GKS. La tarea no está utilizando actualmente el GKS
- Esperando GKS. La tarea está esperando que el planificador permita su ejecución.
- Bloqueada. La tarea ha realizado una primitiva de entrada bloqueante y está esperando que finalice la operación.
- Ejecutando GKS. Es el estado de la única tarea que actualmente está accediendo a GKS.

En la ejecución de una tarea que utiliza GKS se pueden distinguir intervalos en los que la tarea no accede a GKS (estado NO GKS) e intervalos de acceso al GKS. Se incluyen dos nuevas primitivas GKS_START Y GKS_END que permiten a la tarea informar al planificador cuando comienza un intervalo de acceso y cuando termina. Con el GKS_START se pasa como parámetros el identificador de la tarea y la prioridad de acceso al GKS (prioridad para el planificador, no para Ada).

El planificador utiliza una política expulsiva con varios niveles de prioridad asignando rodajas de tiempo entre las tareas de la misma prioridad. Se ha diseñado como una unidad genérica para poder experimentar con los parámetros de la planificación.

En la figura 7.3 se presenta un esquema de la solución propuesta. Teniendo presente dicho esquema vamos a exponer algunas características de esta solución:

- El planificador incluye puntos de entrada para que otros módulos le informen de los siguientes eventos: Una tarea inicia un intervalo GKS (GKS_START); la tarea que está accediendo a GKS termina su intervalo (GKS_END); la tarea ejecutando GKS ha iniciado una petición de entrada bloqueante; una tarea se desbloquea debido a que ha terminado la operación bloqueante correspondiente (estos dos últimos eventos los notifica el gestor de entrada). Incluye otros dos puntos de entrada para realizar la planificación: El primero para detener la tarea

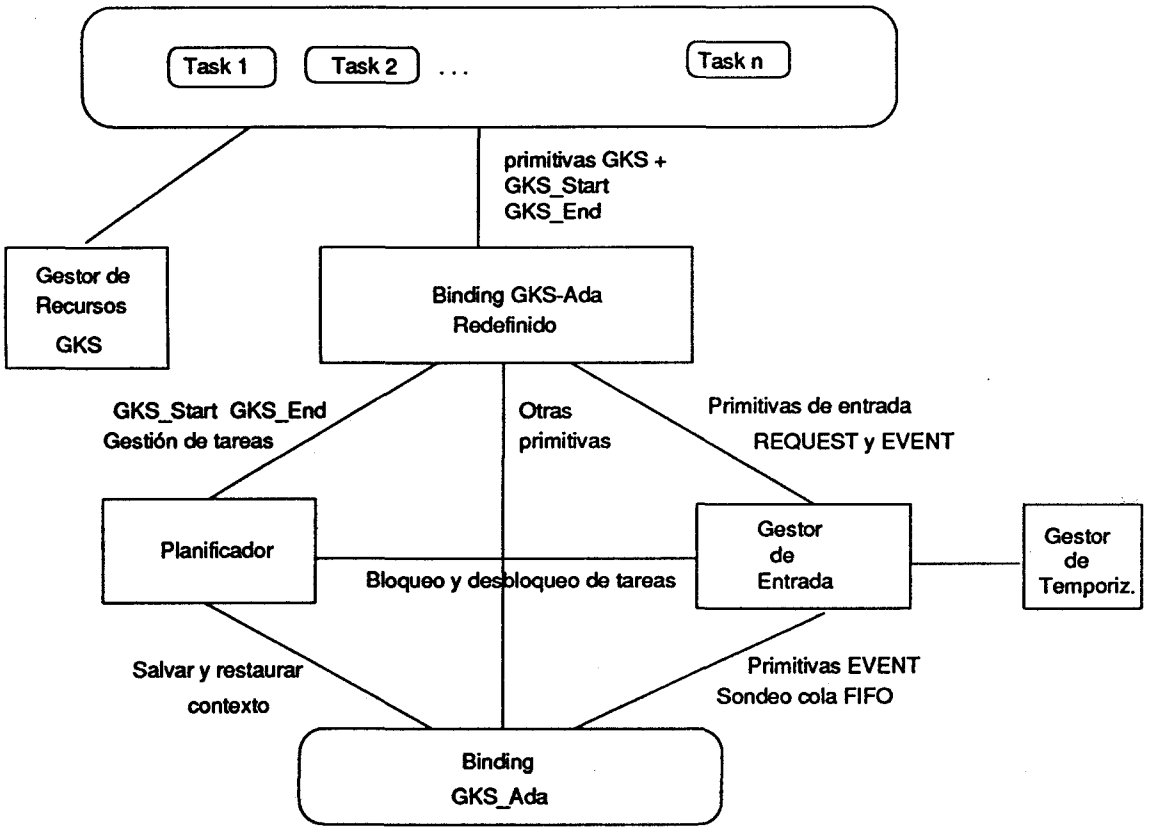

Fig. 7.3 Esquema global del sistema

que accedía a GKS y otro, una familia de puntos de entrada, para seleccionar la nueva tarea.

El planificador debe incluir, además de las colas de planificación, una cola para las tareas bloqueadas.

- Las primitivas del *binding* redefinido, además de invocar la correspondiente primitiva, interrogan al planificador para saber si existe un cambio de contexto o la tarea puede continuar accediendo a GKS. No es preciso incluir un identificador de la tarea en cada primitiva, sólo es necesario en la primitiva (GKS_START).

- El gestor de entrada se encarga de gestionar los modos REQUEST y EVENT. Cuando recibe una primitiva bloqueante, informa al planificador del evento. Por otro lado, inspecciona periódicamente la cola FIFO, de forma que cuando recoge una entrada para una petición REQUEST pendiente (o una EVENT temporizada) informa al planificador para que desbloquee la tarea. Las entradas correspondientes a dispositivos que la tarea puso en modo EVENT se almacenan en colas FIFO que mantiene el gestor, una por cada tarea.

- Existen otros dos módulos, el gestor de recursos y el de temporizaciones, que permiten, respectivamente, la asignación de identificadores a los diferentes elementos GKS y a las distintas tareas, y el manejo de las temporizaciones que permiten simular los plazos de las primitivas GKS_AWAIT_EVENT.

Hay que destacar que se ha mantenido el *binding* original incluyendo únicamente dos primitivas (GKS_START y GKS_END).

En cuanto a la eficiencia y al tiempo de respuesta a las entradas (la cola de entradas se sondea periódicamente), depende tanto de las características de la aplicación como de los valores asignados a los diferentes parámetros ajustables del sistema.

7.6 Sumario del capítulo

En este capítulo se ha planteado un ejemplo práctico, el acceso al estándar gráfico GKS desde un programa Ada, que presenta algunos de los problemas analizados en el capítulo anterior.

El acceso simultáneo e independiente a GKS de las tareas de un programa presenta dos tipos de problemas. Por un lado, algunas operaciones de entrada de GKS, las correspondientes al modo REQUEST y al modo EVENT con temporización, son bloqueantes. Por otro lado, el resultado de una primitiva invocada por una tarea depende del estado en que se encuentre el GKS en ese instante, por lo tanto, va a depender de las primitivas que hayan invocado anteriormente las restantes tareas del programa.

Para resolver el primer problema es necesario simular la entrada bloqueante mediante un tipo de entrada no bloqueante, el modo EVENT sin temporización, dejando únicamente bloqueada la tarea que invocó la operación.

En el segundo caso, se necesita proporcionar acceso exclusivo al GKS pero, además, hay que asegurar que el estado de GKS es el adecuado antes de que una tarea invoque una primitiva. Para ello se deberá salvar y restaurar el estado GKS según las diferentes tareas accedan al mismo. Por razones de eficiencia, es necesario que estas operaciones no se realicen por cada primitiva y, por lo tanto, se le permitirá a una tarea invocar una serie de primitivas antes de salvar su estado y dejar acceder a otra tarea.

La solución adoptada está basada en utilizar un planificador que se encarga de controlar el acceso de las tareas del programa a GKS. El planificador se ha implementado de una forma genérica lo que permite modificar la política del mismo. Una determinada tarea, una vez que gana acceso a GKS, continuará ejecutando primitivas hasta que, o bien, realice una operación bloqueante, o bien, el planificador decida expulsarla al existir una tarea de mayor prioridad que quiere acceder al GKS o al haberse terminado la rodaja de tiempo correspondiente. En ambos casos, se expulsará dicha tarea salvando su estado y se activará otra tarea restaurando el estado de la misma.

8

Conclusiones

La principal conclusión de este trabajo es que, frente a algunas opiniones contrarias, los lenguajes con paralelismo explícito pueden considerarse como buenas herramientas para la programación de máquinas paralelas, cuando se evalúan con respecto a la transportabilidad de los programas escritos con este tipo de lenguajes a máquinas con muy diferentes prestaciones.

La solución propuesta en esta tesis para resolver los problemas de transportabilidad de los programas con paralelismo explícito, es la utilización del exceso de paralelismo, esto es, el uso de mayor paralelismo en el programa que el existente en la máquina en la que se ejecuta, para esconder la latencia en la comunicación. Con esta técnica, si el paralelismo del programa es suficientemente mayor que el de la máquina, se podrá esconder totalmente la latencia en las comunicaciones siendo, por lo tanto, la utilización de la máquina independiente de la latencia.

Hemos desarrollado un análisis cuantitativo sobre el uso del exceso de paralelismo con este tipo de lenguajes que nos ha permitido obtener algunos resultados de interés. Entre ellos se puede destacar la formulación de una heurística que permite, siempre que se cumplan las condiciones necesarias, estimar cuando un programa se ejecutará eficientemente en una determinada máquina. Esta heurística se basa en ca-

racterizar a los programas y a las máquinas paralelos mediante el producto del número de procesos por el tiempo entre comunicaciones en cada proceso y el producto del número de procesadores por la latencia, respectivamente. El programador deberá codificar el programa con el máximo paralelismo posible y, en caso de que no se ejecute eficientemente en una determinada máquina, no necesitará modificar el programa sino utilizar menos procesadores.

Del análisis cuantitativo se ha identificado la necesidad, sobre todo con programas de granularidad muy fina, de que la máquina proporcione soporte eficiente en los siguientes aspectos: Gestión de procesos, red de interconexión con una elevada tasa de transferencias aunque la latencia pueda ser alta, y transferencias de ciclo partido evitando que el procesador se bloquee mientras dura una transferencia. Se han evaluado, con respecto a estas necesidades, algunas propuestas experimentales como el procesador P-RISC.

Otro aspecto de esta tesis que es conveniente resaltar, es el desarrollo de un modelo jerárquico de paralelismo. Este nuevo modelo ha servido de marco de referencia para el resto del trabajo, y ha permitido realizar un análisis genérico de la implementación de lenguajes paralelos. Este análisis puede aplicarse tanto a la implementación sobre el sistema operativo como directamente sobre la máquina física.

En cuanto al lenguaje Ada, del análisis de las características de este lenguaje que pueden afectar a la eficiencia en la ejecución, se han obtenido las siguientes conclusiones:

- El uso de copias locales de las variables compartidas, como alternativa al exceso de paralelismo, presenta algunos aspectos problemáticos. Se han analizado distintas alternativas para realizar la gestión de las copias locales. En primer lugar, la gestión estática obliga a incluir nuevos puntos de sincronización que degradan el rendimiento. La gestión dinámica sin apoyo hardware, en cambio, elimina la necesidad de la mayoría de estos puntos de sincronización, aunque mantiene los asociados a la invocación y retorno de subprogramas. La solución que consideramos más adecuada es la utilización de memorias cache con esquemas de cohe-

rencia software. Hemos desarrollado algunos algoritmos para mantener la coherencia de la cache orientados a las características específicas de Ada.

- El mantenimiento de las reglas de ámbito, y la consiguiente gestión del *cactus stack*, no presenta, en nuestra opinión, problemas de eficiencia aunque complican considerablemente el entorno de ejecución Ada.

- El mecanismo de terminación de tareas es un aspecto que afecta a la eficiencia de la implementación. Hemos propuesto modelos de terminación de tareas alternativos cuya implementación sea menos problemática.

El estudio de la problemática específica de implementar Ada sobre el sistema operativo nos ha permitido identificar algunos problemas, entre ellos se puede destacar al que hemos denominado problema de los bloqueos anidados. Este problema se presenta, generalmente, cuando un tarea interacciona con el entorno externo provocando el bloqueo del correspondiente proceso del sistema operativo. Hemos planteado dos alternativas para eliminar las repercusiones de este problema. Por un lado, asociar cada tarea a un proceso del sistema. Esta estrategia puede ser factible cuando el sistema operativo proporciona procesos "ligeros" (*ligth-weigth process*). Por otro lado, simular el servicio bloqueante a partir de servicios no bloqueantes (asíncronos), de forma que el bloqueo sólo afecte a la tarea que invocó el servicio.

Por último, el estudio de estos problemas en un caso práctico, el acceso concurrente al estándar gráfico GKS desde Ada, junto con la implementación de una solución, nos ha mostrado que, en algunas ocasiones, la solución eficiente de los mismos puede ser bastante compleja. En este caso ha sido necesario desarrollar un planificador que controle el acceso de las tareas a GKS y se encargue de gestionar la entrada gráfica. El carácter genérico y ajustable del planificador implementado nos ha permitido experimentar con diferentes políticas de planificación.

Futuras líneas de trabajo

Las características de este trabajo, en el que el paralelismo explícito es el hilo conductor pero en el que se analizan temas bastante diversos, hacen que no exista una línea de trabajo principal con algunas ramificaciones, sino varias líneas relativamente independientes. Algunas cuestiones que han quedado abiertas y sobre las que se puede profundizar son las siguientes:

- El modelo jerárquico de paralelismo planteado se centra en el paralelismo asíncrono. En el trabajo se han expuesto algunas breves consideraciones sobre el paralelismo síncrono reflejando su carácter jerárquico. Sería interesante realizar un estudio más detallado de este tipo de paralelismo que permita su inclusión en el modelo jerárquico. Se analizarían aspectos como, por ejemplo, la existencia o no de memoria compartida entre las unidades de proceso que ejecutan síncronamente.

- Un factor clave para ejecutar eficientemente programas con granularidad fina es que la red de interconexión, ya sea procesador-procesador o procesador-memoria, proporcione una elevada tasa de transferencia aunque la latencia sea alta. El diseño de redes que tengan estas características es un tema de gran interés.

- Para ejecutar eficientemente programas con granularidad fina, es necesario que el procesador proporcione soporte directo a la gestión de procesos. En el trabajo se ha descrito un procesador de este tipo, el P-RISC que ejecutaba simultáneamente instrucciones de diferentes procesos, pero que no ejecutaba la instrucción de un proceso hasta que no terminaba completamente la anterior. Se pueden plantear otras arquitecturas que sigan diferentes alternativas, como ejecutar instrucciones del mismo proceso hasta que se bloquee.

- El análisis de la gestión de las copias locales está basado en el Ada 83. En Ada 9X se ha modificado la definición de las variables compartidas introduciendo

nuevas clases como las variables volátiles. Se debería revisar el análisis realizado para que incluyera estas nuevas características.

- La gestión estática, en tiempo de compilación, de las copias locales implica la inclusión de nuevos puntos de sincronización que disminuyen el rendimiento de este mecanismo. Debido a la sencillez de la solución estática, sería de interés estudiar algoritmos que minimicen esta inclusión.

- La implementación de Ada sobre sistemas operativos con procesos ligeros (*threads*), asociando cada tarea a un proceso, se presenta como una opción muy interesante. Para poder llevar a cabo esta solución de manera eficiente, las características del sistema operativo deben ser relativamente compatibles con las de Ada. Se pueden analizar diferentes sistemas operativos con procesos ligeros, como Mach y Amoeba, para evaluar si son adecuados para este tipo de implementación.

- La solución propuesta para eliminar los problemas en el acceso concurrente a GKS ha sido implementada de forma genérica. Se puede intentar adaptar esta solución a otras situaciones que presenten problemas similares como, por ejemplo, el acceso a una base de datos.

9

Bibliografía

[ADA9Xa] Ada 9X Project Report. *Draft Mapping Document*, Febrero 1991.

[ADA9Xb] Ada 9X Project Report. *Draft Mapping Rationale Document*, Febrero 1991.

[AHU86] Ahuja S. Linda and Friends. *Computer*. Agosto 1986, pp.26-34.

[ALSa] Alsys Ada. Sun Workstations. Versión 4.1.

[ALSb] Multi-Transputer. Alsys Ada Compilation System for the Transputer. Versión 4.3.

[ALV90] Alvarez A. y Gómez F. Informe "Fourth International Workshop on Real Time Ada Issues". *SpAda*, Octubre 1990, pp. 14-27.

[ARD87] Ardo A. Real-Time efficiency of Ada in a Multiprocessor Environment. *Proceedings of the First International Workshop on Real Time Ada Issues. Ada Letters*, Mayo 1987, pp. 40-42.

[ARE88] Arévalo S. Alvarez A. Fault Tolerant Distributed Ada. *Proceeding of the Second International Workshop on Real Time Ada Issues. Ada Letters*, Noviembre 1988, pp. 118-122.

[ART88] *A Catalog of Interface Features and Options for the Ada Run Time Environment*. ARTEWG, ACM, 1988.

[ATK88] Atkinson C. y Goldsack S. Communication Between Ada Programs in DIADEM. *Proceedings of the Second International Workshop on Real Time Ada Issues. Ada Letters*, Noviembre 1988. pp. 86-96.

[BAK85] Baker P. y Ricarddi A. Ada Tasking: From Semantics To Efficient Implementation. *IEEE Software*. Marzo 1985, pp.34-45.

[BAK90] Baker T. Protected Records, Time Management, and Distribution. *Proceedings of the Fourth International Workshop on Real Time Ada Issues. Ada Letters*, 1990, pp. 17-28.

[BAL89] Bal H. et al. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, Septiembre 1989, pp. 261-321.

[BEC89] Beck B. y Olien D.A. Parallel-Programming Process Model. *IEEE Software*, Mayo 1989, pp. 63-72.

[BEN90] Ben Ari M. Signaling from within Interrupt Handlers. *Ada Letters*, Enero 1990.

[BLA90] Black D. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *Computer*, Mayo 1990, pp. 35-43.

[BRY90] Bryan D. Dear Ada. *Ada Letters*, Mayo 1990.

[BUR85] Burns A. *Concurrent programming in Ada.* Cambridge University Press. 1985.

[BUR87] Burns A. y Lister A. y Wellings A. *A Review of ADA Tasking.* Lectures Notes in computer Science. Springer-Verlag. 1987.

[BUR90] Burns A. y Wellings A. *Real Time Systems and their Programming Languages*, Addison-Wesley. 1990.

[CAR88] Carretero J. *Entorno Gráfico Concurrente.* Proyecto Fin de Carrera. Facultad de Informática. Madrid, Julio 1988.

[CAR89a] Carriero N. y Gelernter D. Linda in Context. *Communications of the ACM*, Abril 1989, pp. 444-458.

[CAR89b] Carriero N. y Gelernter D. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, Septiembre 1989, pp. 323-357.

[CHE90] Cheong H. y Veidenbaum A. Compiler-Directed Cache Management im Multiprocessors. *Computer.* Junio 1990, pp. 39-47.

[CHE91] Cheriton D y Goosen H. Paradigm: A Highly Scalable Shared-Memory multicomputer Architecture. *Computer*, Febrero 1991, pp. 33-46.

[CME89] Cmelik R. et al. Experience with Multiple Processor Versions of Concurrent C. *IEEE Transactions on Software Engineering.* Vol.15, No.3, Marzo 1989. pp. 335-344.

[COU88] Coulouris G. y Dollimore J. *Distributed Systems.* Addison-Wesley. 1988.

[DEN90] Denning P. The Science of Computing About Time. *American Scientist*, Julio 1990, pp. 303-306.

[DIJ68] Dijkstra E. Cooperating sequential processes. *In Programming Languages*, 1968.

[DOB90] Dobbing B. Distributed Ada: A suggested Solution for Ada 9X. *Proceedings of the Fourth International Workshop on Real Time Ada Issues. Ada Letters*, 1990, pp. 94-102.

[DUN90] Duncan R. A survey of parallel computer architectures. *Computer*, Febrero 1990, pp. 5-16.

[FAN84] Fantechi A. Interfacing with real environments from Ada programs. *Ada Letters*, Vol.4, No.2. 1984.

[FEI90] Feitelson G. y Rudolph L. Distributed Hierarchical control for Parallel Processing. *Computer*, Mayo 1990, pp. 65-76

[FIS86] Fisher d. y Weatherly R. Issues in the Design of a Distributed Operating System for Ada. *Computer*, Mayo 1986, pp. 38-47.

[FLY87] Flynn S. el al. The Efficient Termination of Ada taks in a Multiprocessor Environment. *Ada Letters*, Noviembre 1987, pp. 55-76.

[FLY89] Flynn s. el al. A Storage Model for Ada on Hierarchical-Memory Multiprocessors. *Ada: the Design Choice Proceedings of the Ada-Europe International Conference.* 1989.

[GAR90a] Gargaro A. el al. Virtual Nodes/Distributed Systems Working Group . *Proceedings of the Third International Workshop on Real Time Ada Issues. Ada Letters*, 1990, pp. 66-77.

[GAR90b] Gargaro A. el al. Adapting Ada for Distribution and Fault Tolerance. *Proceedings of the Fourth International Workshop on Real Time Ada Issues. Ada Letters*, 1990, pp. 111-117.

[GEH88] Gehani N. y Roome W. Rendezvous Facilities: Concurrent C and the Ada Language. *IEEE Transactions on Software Engineering.* Noviembre 1988, pp. 1546-1552.

[GEH89] Gehani N. y Roome W. *The Concurrent C Programming Language.* Silicon Press. 1989.

[GEHR88] Gehringer E. el al. A Survey of Commercial Parallel Processors. *Computer Architecture News*, Septiembre 1988.

[GKS85] International Organisation for Standardisation (ISO), *Graphical Kernel System*, ISO 7942, 1985.

[GKS88] International Organisation for Standardisation (ISO), *GKS-Ada Binding*, ISO 8651-3, 1988.

[GOL90] Goldsack S. y Atkinson C. An Object Oriented Approach to Virtual Nodes: Are Package Types an Answeer? *Proceedings of the Third International Workshop on Real Time Ada Issues. Ada Letters*, 1990, pp. 78-84

[HOP87] Hopgood F.R.A. et al. *Introduction to the Graphical Kernel System.* Academic Press. 1987.

[HUT88] Hutcheon A.D. y Wellings A.J. Supporting Ada in a Distributed Environment. *Proceedings of the Second International Workshop on Real Time Ada Issues. Ada Letters*, Vol.8, No.7. 1988.

[HUT89] Hutcheon A.y Wellings A. Elaboration and Termination of Distributed Ada Programs. *Ada: the Design Choice Proceedings of the Ada-Europe International Conference.* 1989.

[IAN88] Iannucci R. Toward a Dataflow/von Neumann Hybrid Architecture, Proc. *15th Annual Intl. Symp. on Computer Architecture*, Junio 1988, pp.131-139.

[ICH86] Ichbiah j. et al. *Rationale for the Design of the Ada Programming language.* Silicon Press, 1986.

[JES82] Jessop W. Ada Packages and Distributed Systems. *Sigplan Notices*, 1982, pp. 28-36.

[JHA89a] Jha R. et al. An Implementation Supporting Distributed Execution of Partitioned Ada Programs. *Ada Letters*, Vol.9, No.1. Enero 1989.

[JHA89b] Jha R. et al. Ada Program Partitioning Language: A Notation for Distributing Ada Programs. *IEEE Transactions on Software Engineering*, Marzo 1989.

[KEE85] Keefe B. et al. *PULSE: An Ada-based Distributed Operating System.* Academic Press. 1985.

[KNI87] Knight J. y Urquhart J. On the implementation and Use of Ada on Fault-Tolerant Distributed Systems. *IEEE Transactions on Software Engineering*, Mayo 1987.

[KRI88] Krishnan P et al. Implementation of Task Types in Distributed Ada. *Proceedings of the Second International Workshop on Real Time Ada Issues. Ada Letters*, Noviembre 1988, pp. 104-107.

[MER] Meridian AdaVantage Compiler, Versión 4.0.

[MUD87] Mudge T. Units of Distribution for Distributed Ada. *Proceedings of the First International Workshop on Real Time Ada Issues. Ada Letters*, Nobiembre 1987, pp. 64-66.

[MUN86] Mundie D. y Fisher D. Parallel Processing in Ada. *Computer*, Agosto 1986. pp. 20-25.

[MRL] Reference Manual for the Ada Programming Language Ansi/Mil-STD-1815A-1983.

[NIK89] Nikhil R. Can Dataflow subsume von Neumann computing? *Proc. 16th Annual Intl. Symp. on Computer Architecture*, Junio 1989, pp. 262-272.

[PAZ90] Pazy O. Problems With Pthreads and Ada. *Proceedings of the Fourth International Workshop on Real Time Ada Issues. Ada Letters*, 1990, pp. 133-140.

[PER87] Pérez F. *Modelo para la Implementación de Sistemas Abiertos. Aplicación a un Sistema de Mensajería Basado en la Recomendación X-400.* Proyecto Fin de Carrera. Facultad de Informática. Madrid, Julio 1987.

[PER89] Pérez F. et al. Ada Mechanisms To Obtain Concurrency In GKS. *Proceedings of the Ada-Europe International Conference.* Cambridge University Press. Madrid 1989.

[PET85] Peterson J. y Silberschatz A. *Operating System Concepts.* Addison-Wesley, 1985.

[QUI90] Quinn M. y Hatcher P. Data-Parallel Programming on Multicomputers. *IEEE Software.* Septiembre 1990. pp. 69-75.

[SAR89] Sarkar V. *Partitionning and Scheduling Parallel Programs for Multiprocessors.* Pitman Publishing, 1989.

[SKI90] Skillicorn D. Architecture-Independent Parallel Computation. *Computer*, Diciembre 1990, pp. 38-50.

[STE90] Stenström P. A Survey of Cache Coherence Schemes Multiprocessors. *Computer.* Junio 1990, pp. 12-24.

[TAN84] Tanembaum A. *Structured Computer Organization.* Prentice-Hall, 1984.

[TED84] Tedd M. el al. *Ada for Microprocessors.* Cambridge Uneversity Press, 1984.

[VAL90] Valiant L. A. Bridging Model for Parallel Computation. *Communications of the ACM*, Agosto 1990, pp. 103-111.

[VAX] *Vax Ada Programmer's Run-time Reference Manual.* Versión 1.0.

[VEG90] Vega L. *GKS-Ada Binding*. Proyecto Fin de Carrera. Facultad de Informática. Madrid, 1990.

[VER] VERDIX Ada Development System. Versión 5.5.

[VOL87] Volz R.A. y Mudge T.N. Timing Issues in the Distributed Execution of Ada Programs. *IEEE Transations on Computers*, Vol. C-36, No.4, Abril 1987, pp. 449-459.

[VOL89] Volz R.A. et al. Translation and Execution of Distributed Ada Programs: Is it Still Ada? *IEEE Transations on Software Engineering*. Vol.15. No.3. Marzo 1989. pp. 281-292.

[VOL90] Volz A. Virtual Nodes and Units Of Distribution for Distributed Ada. *Proceedings of the Third International Workshop on Real Time Ada Issues. Ada Letters*, Noviembre 1990, 85-96.

[ZAM90] Zamorano J. et al. Using GKS Concurrently: A Practical solution. *Computer Graphics Forum*, Septiembre 1990, pp. 239-244.